

Python 精要教程

Table of Contents

Python 精要教程

来源：随笔分类 - Python

作者：Marlowes

第一章 快速改造：基础知识

> 来源：http://www.cnblogs.com/Marlowes/p/5280405.html
>
> 作者：Marlowes

1.1 安装Python （略……）

安装Python教程网上能找到很多，这里我不想手打了......

1.2 交互式解释器

当启动Python的时候，会出现和下面相似的提示：

```
Python 2.7.11 (v2.7.11:6d1b6a68f775, Dec  5 2015, 20:40:30) [MSC v.1500 64 bit (AMD64)
] on win32
Type "help", "copyright", "credits" or "license" for more information.
```

注：不同的版本的提示和错误信息可能会有所不同。

进入Python的交互式解释器之后输入下面的命令查看它是否正常的工作：

```
>>> print "Hello,world!"
```

按下回车后，会得到下面的输出：

```
Hello,world!
```

1.3 算法是什么

首先解释一下什么是计算机程序设计。简单地说，它就是告诉计算机要做什么。计算机可以做很多的事情，但是不太擅长自主思考，程序员要像给小孩喂饭一样告诉它具体的细节，并且使用计算机能够理解的语言——算法。“算法”不过是“步骤”或者“食谱”的另外一种说法——对于如何做某事的一份详细的表述。比如：

> SPAM拌SPAM、SPAM、鸡蛋和SPAM：
>
> 首先，拿一些SPAM；
>
> 然后加入一些SPAM、SPAM和鸡蛋；

如果喜欢吃特别辣的SPAM，再多加点SPAM；

煮到熟为止——每10分钟检查一次。

这个食谱可能不是很有趣，但是它的组成结构还是有些讲究的。它包括一系列按顺序执行的指令。有些指令可以直接完成（“拿一些SPAM”），有些则需要考虑特定的条件（“如果需要特别辣的SPAM”），还有一些则必须重复次数（“每10分钟检查一次”）。

食谱和算法都包括一些材料（对象，物品），以及指令（语句）。本例中，“SPAM”和“鸡蛋”就是要素，指令则包括添加SPAM、按照给定的时间烹调，等等。

1.4 数字和表达式

交互式Python解释器可以当作非常强大的计算器使用，如以下的例子：

```
>>> 256868 + 684681
941549
```

以上是非常普通的功能。在绝大多数情况下，常用算术运算符的功能和计算器的相同。这里有个潜在的陷阱，就是整数除法（在3.0版本之前的Python是这样的）。

```
>>> 1 / 2
0
```

从上面的例子可以看出，一个整数（无小数部分的数）被另一个整数除，计算结果的小数部分被截除了，只留下整数部分。有些时候，这个功能很有用，但通常人们只需要普通的除法。有两个有效的解决方案：要么用实数（包含小数点的数），而不是整数进行运算，要么让Python改变除法的执行方式。

实数在Python中被称为浮点数（Float，或者Float-point Number），如果参与除法的两个数中有一个数为浮点数，则运算结果亦为浮点数：

```
>>> 1.0 / 2.0
0.5
>>> 1 / 2.0
0.5
>>> 1 / 2.
0.5
```

如果希望Python只执行普通的除法，那么可以在程序前加上一下语句，或者直接在解释器里面执行它：

```
>>> from __future__ import division
```

还有另一个方法，如果通过命令行运行Python，可以使用命令开关`-Qnew`。使用上述两种方法，就可以只执行普通的除法运算：

```
>>> 1/2
0.5
```

当然，单斜线不再用作前面提到的整除了，但是Python提供了另外一个用于实现整除的操作符——双斜线：

```
>>> 1 // 2
0
```

就算是浮点数，双斜线也会执行整除：

```
>>> 1.0 // 2.0
0.0
```

现在，已经了解基本的算数运算符了（加、减、乘、除）。除此之外，还有一个非常有用的运算符：

```
>>> 1 % 2
1
```

这是取余（模除）运算符——`x % y`的结果为`x`除以`y`的余数。下面是另外一些例子：

```
>>> 10 / 3
3
>>> 10 % 3
1
>>> 9 / 3
3
>>> 9 % 3 0
>>> 2.75 % 0.5
0.25
```

这里`10/3`得`3`是因为结果被向下取整了。而`3x3=9`，所以相应的余数就是`1`了。在计算`9/3`时，结果就是`3`，没有小数部分可供截除，因此，余数就是`0`了。如果要进行一些类似文章前面菜谱所述“每10分钟”检查一次的操作，那么，取余运算就非常有用了，直接检查时间`10%`的结果是否为`0`即可。从上述最后一个例子可以看到，取余运算符对浮点数也同样适用。

最后一个运算符就是幂（乘方）运算符：

```
>>> 2 ** 3
8
>>> -3 ** 2
-9
>>> (-3) ** 2
9
```

注：幂运算符比取反（一元减运算符）的优先级要高，所以`-3**2`等同于`-(3**2)`。如果想计算`(-3)**2`，就需要显示说明。

1.4.1 长整数

Python可以处理非常大的整数：

```
>>> 1000000000000000000000
1000000000000000000000L
```

可以看到数字后面自动加上了一个L

普通的整数不能大于`2 147 483 647`（也不能小于`-2 147 483 648`），如果需要更大的数，可以使用长整数。长整数的书写方法和普通整数一样，但是结尾有个`L`。（理论上，用小写的`l`也可以，但是小写的l看起来太像`1`，所以建议不要用小写。）

在前面的例子中，Python把整数转换为了长整数，但是我们自己也可以完成：

```
>>> 1000000000000000000000L
1000000000000000000000L
```

当然，这只是在不能处理大数的旧版Python中很有用。

也可以对这些庞大的数字进行运算，例如：

```
>>> 1651658468354135451L * 26546843513654354341L + 1846846746843
43846318897399266344521301723330037 3L
```

正如所看到的那样，长整数和普通整数可以混合使用。在绝大多数情况下，无需担心长整数和整数的区别，除非需要进行类型检查。

1.4.2 十六进制和八进制

在Python中，十六进制数应该像下面这样书写：

```
>>> 0xAF
175
```

而八进制数则是：

```
>>> 010
8
```

十六进制和八进制数的首位数字都是0

1.5 变量

变量（variable）是另外一个需要熟知的概念。Python中的变量很好理解。变量基本上就是代表（或者引用）某值的名字。举例来说，如果希望用名字 `x` 代表 `3` ，只需要执行下面的语句即可：

```
>>> x = 3
```

这样的操作成为赋值（assignment），数值3被赋值给了变量 `x` 。或者说：将变量x绑定到了值（或者对象） `3` 上面。在变量被赋值之后，就可以在表达式中使用变量。

```
>>> x * 2
6
```

注意，在使用变量之前，需要对其赋值。毕竟不代表任何值的变量没有什么意义。

注：变量名可以包括字母、数字和下划线（` ）。变量不能以数字开头，所以 Plan9 是合法变量名，而 9Plan`不是。_

1.6 语句

到现在为止，我们一直都在讲述表达式，也就是“食谱”的“材料”。那么，语句（也就是指令）是什么呢？

刚才已经介绍了两类语句： `print` 语句和赋值语句。那么语句和表达式之间有什么区别呢？表达式就是某件事情，而语句是做某件事情（即告诉计算机做什么）。比如 `2*2` 是 `4` ，而 `print` 2*2 则是打印 4`。那么区别在哪呢？毕竟，它们的行为非常相似。请看下面的例子：

```
>>> 2 * 2
4
>>> print 2 * 2
4
```

如果在交互式解释器中执行上述两行代码，结果是一样的。但这只是因为交互式解释器总是把所有表达式的值打印出来而已（都使用了相同的 `repr` 函数对结果进行呈现，详细参见1.11.3节）。一般情况下，Python并不会那样做。在程序中编写类似 `2*2` 这样的表达式并不

能做什么有趣的事情（它只计算了 `2*2` 的结果，但是结果并不会在某处保存或显示给用户，它对运算本身之外的东西没有任何的副作用）。另一方面，编写 `print 2*2` 则会打印出 `4` 。

语句和表达式之间的区别在赋值时会表现的更加明显一些。因为语句不是表达式，所以没有值可供交互式解释器打印出来：

```
>>> x = 3
>>>
```

可以看到，下面立刻出现了新的提示符。但是，有些东西已经变化了， `x` 现在绑定给了值 `3` 。

这也是能定义语句的一般性特征：它们改变了事务。比如，赋值语句改变了变量， `print` 语句改变了屏幕显示的内容。

赋值语句可能是任何计算机程序设计语言中最重要的语句类型，尽管现在还难以说清它们的重要性。变量就像临时的“存储器”（就像烹饪食谱中的锅碗瓢盆一样。但是值并没有保存在变量中——它们保存在计算机内存的深处，被变量引用。随着本书内容的深入，你会对此越来越清楚：多个变量可以引用同一个值。），它的强大之处就在于，在操作变量的时候并不需要知道它们存储了什么值。比如，即使不知道x和y的值到底是多少，也会知道 `x*y` 的结果就是 `x` 和 `y` 的乘积。所以，可以在程序中通过多种方法来使用变量，而不需要知道在程序运行的时候，最终存储（或引用）的值到底是什么。

1.7 获取用户输入

我们在编写程序的时候，并不需要知道变量的具体值。当然，解释器最终还是得知道变量的值。可是，它怎么会知道我们都不知道的事呢？解释器只知道我们告诉它的内容，对吧？不一定。

事实上，我们通常编写程序让别人用，我们无法预测用户会给程序提供什么样的值。那么，看看非常有用的 `input` 函数吧：

```
>>> input("You this year many big?: ")
You this year many big?: 18
18
```

在这里，交互式解释器执行了第一行的 `input(...)` 语句。它打印出了字符串 `"You this year many big?:"` ，并以此作为新的提示符，输入 `18` 然后按下回车。 `input` 语句的结果值就是我输入的数字，它自动在最后一行被打印出来。这个例子确实不太有用，但是请接着看下面的内容：

```
>>> x = input("x: ")
x: 16
>>> y = input("y: ")
y: 31
>>> print x * y
496
```

Python提示符(`>>>`)后面的语句可以算作一个完整程序的组成部分了，输入的值(`16` 和 `31`)有用户提供，而程序就会打印出输入的两个数的乘积 `496` 。在编写程序的时候，并不需要知道用户输入的数是多少，对吧？

注：这种做法非常有用，因为你可以将程序存为单独的文件，以便让其他用户可以直接执行。

管窥：if语句

`if` 语句可以让程序在给定条件为真的情况下执行某些操作（执行另外的语句）。一类条件是使用相等运算符 `==` 进行相等性测试。是两个等号，一个等号是用来赋值的。

可以简单地把这个条件放在if后面，然后用冒号将其和后面的语句隔开：

```
>>> if 1 == 2:
        print "One equals two"
        ...
>>> if 1 == 1:
        print "One equals one"
        ...
One equals one
>>>
```

可以看到，当条件为假(`False`)的时候，什么都没有发生；当条件为真(`True`)的时候，后面的语句(本例中为 `print` 语句)被执行。注意，如果在交互式解释器内使用 `if` 语句，需要按两次回车， `if` 语句才能执行。(第五章会对其原因进行说明。)

所以，如果变量 `time` 绑定到当前时间的分钟数上，那么可以使用下面的语句检查是不是“到了整点”。

```
>>> if time % 60 == 0:
        print "On the hour!"
```

1.8 函数

在1.4节中曾经介绍过使用幂运算符(`**`)来计算乘方。事实上，可以用一个函数来代替这个运算符，这个函数就是pow：

```
>>> 2 ** 3
8
>>> pow(2, 3)
8
```

函数就像小型程序一样，可以用来实现特定的功能。Python有很多函数，它们能做很奇妙的事情。你也可以自己定义函数(后面会对此展开讲述)。因此，我们通常把 `pow` 等标准函数称为内建函数。

上例中我使用函数的方式叫做调用函数。你可以给它提供参数(本例中的2和3)。它会返回值给用户。因为它返回了值，函数调用也可以简单看作另外一类表达式，就像在本章前面讨论的算数表达式一样(如果忽略了返回值，函数调用也可以看成语句)。事实上，可以结合使用函数调用和运算符来创建更复杂的表达式：

```
>>> 10 + pow(2, 3 * 5) / 3.0
10932.666666666666
```

还有很多像这样的内建函数可以用于数值表达式。比如使用 `abs` 函数可以得到数的绝对值， `round` 函数则会把浮点数四舍五入为最接近的整数值：

```
>>> abs(-10) 10
>>> round(1.0 / 2.0)
1.0
```

注意最后两个表达式的区别。整数除法总是会截除结果的小数部分，而 `round` 函数则会将结果四舍五入为最接近的整数。但是如果想将给定的数值向下取整为某个特定的整数呢？比如一个人的年龄是32.9岁——但是想把它取整为32，因为她还没到33岁 。Python有实现这样功能的函数(称为 `floor`)，但是不能直接使用它。与其他很多有用的函数一样，你可以在某个模块中找到 `floor` 函数。

1.9 模块

可以把模块想象成导入到Python以增强其功能的扩展。需要使用特殊的命令 `import` 来导入模块。前面内容提到 `floor` 函数就在名为 `math` 的模块中：

```
>>> import math
>>> math.floor(32.9)
32.0
```

注意它是怎么起作用的：用import导入了模块，然后按照“模块.函数”的格式使用这个模块的函数。

如果想把年龄转换为整数(`32`)而不是浮点数(`32.0`)，可以使用 `int` 函数。(`int` 函数/类型把参数转换成整数时会自动向下取整，所以在转换过程中， `math.floor` 是多余的，可以直接使用 `int(32.9)`)

```
>>> int(math.floor(32.9))
32
```

注：还有类似的函数可以将输入数转换为其他类型(比如 `long` 和 `float`)。事实上，他并不完全是普通的函数——它们是类型对象(`type object`)。后面，我将会对类型进行详述。
与 `floor` 相对的函数是 `ceil` (*ceiling*的简写)，可以将给定的数值转换成为大于或等于它的最小整数。

在确定自己不会导入多个同名函数(从不同模块导入)的情况下，你可能希望不要在每次调用函数的时候都写上模块的名字。那么，可以使用 `import` 命令的另外一种形式：

```
>>> from math import sqrt
>>> sqrt(9)
3.0
```

在使用了 `from 模块 import 函数` 这种形式的 `import` 命令之后，就可以直接使用函数，而不需要模块名作为前缀。

注：事实上，可以用变量来引用函数(或者*Python*之中大多数的对象)。比如，通过 `foo = math.sqrt` 进行赋值，然后就可以使用 `foo` 来计算平方根了： `foo(4)` 的结果为 `2.0` 。

1.9.1 `cmath` 和复数

`sqrt` 函数用于计算一个数的平方根。看看如果给它一个负数作为参数会如何：

```
>>> from math import sqrt >>> sqrt(-1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  ValueError: math domain error
```

或者，在其他平台会有以下结果：

```
>>> sqrt(-1)

nan
```

注： `nan` 是一个特殊值的简写，意思是*“not a number” (*非数值*)*。

这也情有可原，不能求负数的平方根。真的不可以么？其实可以：负数的平方根是虚数(这是标准的数学概念，如果感觉有些绕不过弯来，跳过即可)。那么为什么不能使用 `sqrt` ？因为它只能处理浮点数，而虚数(以及复数，即实数和虚数之和)是完全不同的。因此，它们由另一个叫做 `cmath` (即complex math，复数)的模块来处理。

```
>>> import cmath
>>> cmath.sqrt(-1)
1j
```

注意，我在这里并没有使用 `from...import...` 语句。因为一旦使用了这个语句，就没法使用普通的 `sqrt` 函数了。这类命名冲突可能很隐蔽，因此，除非真的需要 `from` 这个形式的模块导入语句，否则应该坚持使用普通的 `import` 。

`1j` 是个虚数，虚数均已 `j` (或者 `J`)结尾，就像长整数使用 `L` 一样。我们在这里不深究复数的理论，只举最后一个例子，来看一下如何使用复数：

```
>>> (1 + 3j) * (9 + 4j)
(-3+31j)
```

可以看到，Python语言本身就提供了对复数的支持。

注：*Python*中没有单独的虚数类型。它们被看做实数部分为*0*的复数。

1.9.2 回到 `__future__`

有传言说Guido van Rossum(Python之父)拥有一架时光机，因为在人们要求增加语言新特性的时候，这个特性通常都已经实现了。当然，我等凡夫俗子是不允许进入这架时光机的。但是Guido很善良，他将时光机的一部分以 `__future__` 这个充满魔力的模块的形式融入了Python。通过它可以导入那些在未来会成为标准Python组成部分的新特性。你已经在1.4节见识过这个模块了，而在本书余下的部分，你还将与它不期而遇。

1.10 保存并执行程序

交互式解释器是Python的强项之一，它让人们能够实时检验解决方案并且用这门语言做一些实验。如果想知道如何使用某些语句，那么就试试看吧！但是，在交互式解释器里面输入的一切都会在它退出的时候丢失。而我们真正想要的是编写自己和他人都能运行的程序。在本节中，将会介绍如何实现这一点。

首先，需要一个文本编辑器，最好是专门用来编程的。如果使用Microsoft Word这样的编辑器(我并不推荐这么做)，那么得保证代码是以纯文本形式保存的。如果已经在用IDLE，那么，很幸运：用File→New Windows方式创建一个新的编辑窗口即可。这样，另外一个窗口就出现了，没有交互式提示符，很好！

先输入以下内容：

```
print "Hello, world!"
```

现在选择File→Save保存程序(其实就是纯文本文件)。要确保将程序保存在一个以后能找到的地方。你应该专门建立一个存放Python项目的目录，还要为自己的程序文件起个有意义的名字。比如 `hello.py` 。文件名以 `.py` 结尾是很重要的。

然后就可以使用Edit→Run或者按下Ctrl+F5键来运行程序了(如果没有用IDLE，请查看下一节有关如何在命令提示符下运行程序的内容)。

你会发现`"Hello, world!"`在解释器窗口内打印出来了，这就是想要的结果。解释器提示符没了(不同的版本会有所差异)，但是可以按下回车键将它找回了(在解释器窗口按回车键)。

接下来，我们对上述脚本进行扩展，如下例所示：

```
name = raw_input("What is your name? ")
print "Hello, " + name + "!"
```

注：不用管*`input`*和*`raw_input`*的区别，稍后，我会进行介绍的。

如果运行这个程序(记得先保存)，应该会在解释器窗口中看到下面的提示：

```
What is your name?
```

输入你的名字(比如`XuHoo`)，然后按下回车键。你将会看到如下内容：

```
Hello, XuHoo!
```

1.10.1 通过命令提示符运行Python脚本

事实上，运行程序的方法有很多。首先，假定打开了DOS窗口或者UNIX中的Shell提示符，并且进入了某个包含Python可执行文件(在Windows中是`python.exe`，而UNIX中则是`python`)的目录，或者包含了这个可执行文件的目录已经放置在环境变量`PATH`中了(仅适用于Windows)。同时假设，上一节的脚本文件(`hello.py`)也在当前目录中。那么，可以在Windows中使用以下命令执行来脚本：

```
C:\> python hello.py
```

或者在UNIX下：

```
$ python hello.py
```

可以看到，命令是一样的，仅仅是系统提示符不同。

注：如果不想跟什么环境变量打交道，可以直接指定*Python*解释器的完整路径。在*Windows*中，可以通过以下命令完成操作：

```
# 根据你的Python版本更改版本号
C:\> C:\Python27\python hello.py
```

1.10.2 让脚本像普通程序一样运行

有些时候希望像运行其他程序(比如Web浏览器、文本编辑器)一样运行Python程序(也叫做脚本)，而不需要显式使用Python解释器。在UNIX中有个标准的实现方法：在脚本首行前面加上 `#!` (叫做pound bang或者shebang)，在其后加上用于解释脚本的程序的绝对路径(在这里，用于解释代码的程序是Python)。即使不太明白其中的原理，如果希望自己的代码能够在UNIX下顺利执行，那么只要把下面的内容放在脚本的首行即可：

```
#!/usr/bin/env python
```

不管Python的二进制文件在哪里，程序都会自动执行。

注：在某些操作系统中，如果安装了最新版的*Python*，同时旧版的*Python*仍然存在*(*因为某些系统程序需要它，所以不能把它卸载*)*，那么在这种情况下， `/usr/bin/env` 技巧就不好用了，因为旧版的*Python*可能会运行程序。因此需要找到新版本*Python*可执行文件*(*可能叫做*python*或*python2)*的具体位置，然后在*pound bang*行中使用完整的路径，如下例所示：

```
#!/usr/bin/python2
```

具体的路径会因系统而异。

在实际运行脚本之前，必须让脚本文件具有可执行的属性(UNIX系统)：

```
$ chmod a+x hello.py
```

现在就能这样运行了(假设当前目录包含在路径中)：

```
$ hello.py
```

注意如果上述操作不起作用的话，试试 `./hello.py` 。即当前的目录(`.`)并不是执行路径的一部分，这样的操作也能够成功。

在Windows系统中，让代码像普通程序一样运行的关键在于后缀名 `.py` 。加入双击上一节保存好的 `hello.py` 文件，如果Python安装正确，那么，一个DOS窗口就会出现，里面有 `"What is your name?"` 提示。

然而，像这样运行程序可能会碰到一个问题：程序运行完毕，窗口也跟着关闭了。也就是说，输入了名字以后，还没来得及看结果，程序窗口就已经关闭了。试着改改代码，在最后加上以下这行代码：

```
raw_input("Press <enter>")
```

这样，在运行程序并且输入名字之后，将会出现一个包含以下内容的DOS窗口：

```
What is your name? XuHoo
Hello, XuHoo!
Press <enter>
```

用户按下回车键以后，窗口就会关闭(因为程序运行结束了)。作为后面内容的预告，现在请你把文件名改为 `hello.pyw` (这是Windows专用的文件类型)，像刚才一样双击。你会发现什么都没有！怎么会这样？在本书后面的内容将会告诉你答案。

1.10.3 注释

井号(`#`)在Python中有些特殊。在代码中输入它的时候，它右边的一切内容都会被忽略(这也是之前Python解释器不会被 `/usr/bin/env` 行“卡住”的原因了)。比如：

```
# 打印圆的周长：
print 2 * pi * radius
```

这里的第一行称为注释。注释是非常有用的，即为了让别人能够更容易理解程序，也为了额你自己回头再看旧代码。据说程序员的第一条戒律就是“汝应注释”(Thou Shalt Comment)(尽管很多刻薄的程序员的座右铭是“如果难写，就该难读”)。程序员应该确保注释说的都是重要的事情，而不是重复代码中显而易见的内容。无用的、多余的注释还不如没有。例如，下例中的注释就不好：

```
# 获得用户名：
user_name = raw_input("What is your name? ")
```

即使没有注释，也应该让代码本身易于理解。幸好，Python是一门出色的语言，它能帮助程序员编写易于理解的程序。

1.11 字符串

那么， `raw_input` 函数和 `"Hello, " + name + "!"` 这些内容到底是什么意思？放下 `raw_input` 函数暂且不表，先来说 `"Hello"` 这个部分。

本章的第一个程序是这样的，很简单：

```
print "Hello, world!"
```

在编程类图书中，习惯上都会以这样一个程序作为开篇——问题是我还没有真正解释它是怎么工作的。前面已经介绍了 `print` 语句的基本知识(随后我会介绍更多相关的内容)，但是 `"Hello, world!"` 是什么呢？是字符串(即“一串字符”)。字符串在几乎所有真实可用的Python程序中都会存在，并且有多种用法，其实最主要的用法就是表示一些文本，比如这个感叹句 `"Hello, world!"` 。

1.11.1 单引号字符串和转义引号

字符串是值，就像数字一样：

```
>>> "Hello, world!"
'Hello, world!'
```

但是，本例中有一个地方可能会让人觉得吃惊：当Python打印出字符串的时候，是用单引号括起来的，但是我们在程序中用的是双引号。这有什么却别吗？事实上，并没有区别。

```
>>> 'Hello, world!'
'Hello, world!'
```

这里也用了单引号，结果是一样的。那么，为什么两个都可以用呢？因为在某些情况下，它们会排上用场：

```
>>> "Let's go!"
"Let's go!"
>>> '"Hello, world!" she said'
'"Hello, world!" she said'
```

在上面的代码中，第一段字符串包含了单引号，这时候就不能用单引号将整个字符串括起来了。如果这么做，解释器会提示错误：

```
>>> 'Let's go!'
  File "<stdin>", line 1
    'Let's go!'
         ^ SyntaxError: invalid syntax
```

在这里字符串为 `'Let'` ，Python并不知道如何处理后面的 `s` (也就是该行余下的内容)。

在第二个字符串中，句子包含了双引号。所以，出于之前所述的原因，就需要用单引号把字符串括起来了。或者，并不一定要这样做，尽管这样做很直观。另外一个选择就是：使用反斜线(`\`)对字符串中的引号进行转义：

```
>>> 'Let\'s go!'
"Let's go!"
```

Python会明白中间的单引号是字符串中的一个字符，而不是字符串的结束标记(即便如此，Python也会在打印字符串的时候在最外层使用双印号)。有的人可能已经猜到，对双引号也可以使用相同的方式转义：

```
>>> "\"Hello, world!\" she said"
'"Hello, world!" she said'
```

像这样转义引号十分有用，有些时候甚至还是必需的。例如，如果希望打印出一个包含单双引号的字符串，不用反斜线的话能怎么办呢？比如字符 `'Let\'s say "Hello, world!"'` ？

注：在本章后面的内容中，将会介绍通过使用长字符串和原始字符串(两者可以联合使用)来减少绝大多数反斜线的使用。

1.11.2 拼接字符串

继续探究刚才的例子，我们可以通过另外一种方式输出同样的字符串：

```
>>> "Let's say" '"Hello, world!"'
'Let\'s say"Hello, world!"'
```

我只是用一个接着另一个的方式写了两个字符串，Python就会自动拼接它们(将它们合为一个字符串)。这种机制用得不多，有时却非常有用。不过，它们只是在同时写下两个字符串时才有效，而且要一个紧接着另一个。否则会出现下面的错误：

```
>>> x = "Hello, "
>>> y = "world!"
>>> x y
  File "<stdin>", line 1 x y
  ^ SyntaxError: invalid syntax
```

换句话说，这仅仅是书写字符串的一种特殊方法，并不是拼接字符串的一般方法。那么，该怎样拼接字符串呢？就像进行加法运算一样：

```
>>> "Hello, " + "world!"
'Hello, world!'
>>> x = "Hello, "
>>> y = "world!"
>>> x + y
'Hello, world!'
```

1.11.3 字符串表示，str和repr

通过前面的例子读者们可能注意到了，所有通过Python打印的字符串还是被引号括起来的。这是因为Python打印值的时候会保持该值在Python代码中的状态，而不是你希望用户所看到的状态。如果使用 `print` 语句，结果就不一样了：

```
>>> "Hello, world!"
'Hello, world!'
>>> 1000000L
1000000L
>>> print "Hello, world!"
Hello, world!
>>> print 1000000L
1000000
```

可以看到，长整型数`1 000 000L`被转换成了数字`1 000 000`，而且在显示给用户的时候也如此。但是当你想知道一个变量的值是多少时，可能会对它是整型还是长整型感兴趣。

我们在这里讨论的实际上是值被转换为字符串的两种机制。可以通过以下两个函数来使用这两种机制：一种是通过`str`函数，它会把值转换为合理形式的字符串，以便用户可以理解；另一种是通过`repr`函数，它会创建一个字符串，以合法的Python表达式的形式来表示值(事实上，`str`和`int`、`long`一样，是一种类型。而`repr`仅仅是函数)。下面是一些例子：

```
>>> print repr("Hello, world!")
'Hello, world!'
>>> print repr(1000000L)
1000000L
>>> print str("Hello, world!")
Hello, world!
>>> print str(1000000L)
1000000
```

`repr(x)`也可以写作`` `x` ``实现(注意，`` ` ``是反引号，而不是单引号)。如果希望打印出一个包含数字的句子，那么反引号就很有用了。比如：

```
>>> temp = 42
>>> print "The temperature is " + temp
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  TypeError: cannot concatenate 'str' and 'int' objects
>>> print "The temperature is " + `temp`
The temperature is 42
```

注：在*Python3.0*中，已经不再使用反引号了。因此，即使在旧的代码中看到了反引号，你也应该坚持使用*`repr`*。

第一个`print`语句并不能工作，那是因为不可以将字符串和数字进行相加。而第二个则可以正常工作，因为我已经通过反引号将`temp`的值转换为字符串`"42"`了。(当然，也可以通过`repr`，也可以得到同样的结果。但是，使用反引号可能更清楚一些。事实上，本例中也可以使用`str`。)

简而言之，`str`、`repr`和反引号是将Python值转换为字符串的三种方法。函数`str`让字符串更易于阅读，而`repr`(和反引号)则把结果字符串转换为合法的Python表达式。

1.11.4 input和raw_input的比较

相信读者已经知道`"Hello, " + name + "!"`是什么意思了，那么，`raw_input`函数怎么用呢？`input`函数不够好吗？让我们试一下。在另外一个脚本文件中输入下面的语句：

```
name = input("What is your name? ")
print "Hello, " + name + "!"
```

看起来这是一个完全合法的程序。但是马上你就会看到，这样是不可行的。尝试运行该程序：

```
What is your name? XuHoo
Traceback (most recent call last):
  File "p.py", line 2, in <module>
name = input("What is your name? ")
  File "<string>", line 1, in <module>
  NameError: name 'XuHoo' is not defined
```

问题在于 `input` 会假设用户输入的是合法的Python表达式(或多或少有些与 `repr` 函数相反的意思)。如果以字符串作为输入的名字，程序运行是没有问题的：

```
What is your name?
"XuHoo" Hello, XuHoo!
```

然而，要求用户带着引号输入他们的名字有点过分，因此，这就需要使用 `raw_input` 函数，它会把所有的输入当做原始数据(raw data)，然后将其放入字符串中：

```
>>> input("Enter a number: ")
Enter a number: 3
3
>>> raw_input("Enter a number: ")
Enter a number: 3
'3'
```

除非对 `input` 有特别的需要，否则应该尽可能使用 `raw_input` 函数。

1.11.5 长字符串、原始字符串和 `Unicode`

在结束本章之前，还会介绍另外两种书写字符串的方法。在需要长达多行的字符串或者包含多种特殊字符的字符串的时候，这些候选的字符串语法就会非常有用。

1.长字符串

如果需要写一个非常非常长的字符串，它需要跨多行，那么，可以使用三个引号代替普通引号：

```
print '''This is a very long string.
It continues here.
And it's not over yet.
"Hello, world!"
Still here.'''
```

也可以使用三个双引号，如 `"""This is a very long string"""` 。注意，因为这种与众不同的引用方式，你可以在字符串之中同时使用单引号和双引号，而不需要使用反斜线进行转义。

注：普通字符串也可以跨行。如果一行之中最后一个字符是反斜线，那么，换行符本身就“转义”了，也就是被忽略了，例如：

```
>>> print "Hello, \
... world!" Hello, world!
```

这个用法也适用于表达式和语句：

```
>>> 1 + 2 + \
... 4 + 5
12
>>> print \
... "Hello, world!"
Hello, world!
```

2.原始字符串

原始字符串对于反斜线并不会特殊对待。在某些情况下这个特性是很有用的(尤其是在书写正则表达式时候，原始字符串就会特别有用)。在普通字符串中，反斜线有特殊的作用：它会转义，让你在字符串中加入通常情况下不能直接加入的内容。例如，换行符可以写为 `\n` ，并可放于字符串中，如下所示：

```
>>> print "Hello, \nworld!"
Hello,
world!
```

这看起来不错，但是有时候，这并非是想要的结果。如果希望在字符串中包含反斜线再加上n怎么办呢？例如，可能需要像DOS路径 `"C:\nowhere"` 这样的字符：

```
>>> path = "C:\nowhere"
>>> path 'C:\nowhere'
```

这看起来是正确的，但是，在打印该字符串的时候就会发现问题了：

```
>>> print path
C:
owhere
```

这并不是期望的结果，那么该怎么办呢？我可以使用反斜线对其本身进行转义：

```
>>> print "C:\\nowhere"
C:\nowhere
```

这看起来不错，但是对于长路径，那么可能需要很多反斜线：

```
>>> path = "C:\\Program Files\\fnord\\foo\\bar\\frozz\\bozz"
```

在这样的情况下，原始字符串就派上用场了。原始字符串不会把反斜线当做特殊字符。在原始字符串中输入的每个字符都会与书写的方式保持一致：

```
>>> print r"C:\nowhere" C:\nowhere
>>> print r"C:\Program Files\fnord\foo\bar\frozz\bozz"
C:\Program Files\fnord\foo\bar\frozz\bozz
```

可以看到，原始字符串以 `r` 开头。看起来可以在原始字符串中放入任何字符，而这种说法也是基本正确的。当然，我们也要像平常一样对引号进行转义，但是，最后输出的字符串包含了转义所用的反斜线：

```
>>> print r'Let\'s go!'
Let\'s go!
```

不能在原始字符串结尾输入反斜线。换句话说，原始字符串最后的一个字符不能是反斜线，除非你对反斜线进行转义(用于转义的反斜线也会成为字符串的一部分)。参照上一个范例，这是一个显而易见的结论。如果最后一个字符(位于结束引号前的那个)是反斜线，Python就不清楚是否应该结束字符串：

```
>>> print r"This is illegal\"
  File "<stdin>", line 1
    print r"This is illegal\"
                           ^ SyntaxError: EOL while scanning string literal
```

好了，这样还是合理的，但是如果希望原始字符只以一个反斜线作为结尾符的话，那该怎么办呢？(例如，DOS路径的最后一个字符有可能是反斜线)好，本节已经告诉了你很多解决此问题的技巧，但本质上就是把反斜线单独作为一个字符串来处理。以下就是一种简单的做法：

```
>>> print r"C:\Program Files\foo\bar" "\\"
C:\Program Files\foo\bar\
```

注：你可以在原始字符串中同时使用单双引号，甚至三引号字符串也可以

3. `Unicode` 字符串

字符串常量的最后一种类型就是 *`Unicode`* 字符串(或者称为 *`Unicode`* 对象，与字符串并不是同一个类型)。如果你不知道什么是 `Unicode` ，那么，可能不需要了解这些内容(如果希望了解更多的信息，可以访问 `Unicode` 的网站 。Python中的普通字符串在内部是以18位的ASCII码形成存储的，而 `Unicode` 字符串则存储为16位的 `Unicode` 字符，这样就能够表示更多的字符集了，包括世界上大多数语言的特殊字符。本节不会详细讲述 `Unicode` 字符串，仅举一下的例子来说明：

```
>>> u"Hello, world!"
u'Hello, world!'
```

可以看到， `Unicode` 字符串使用 `u` 前缀，就像原始字符串使用 `r` 一样。

注：在Python 3.0中，所有字符串都是 `Unicode` 字符串。

1.12 小结

本章讲了非常多的内容。在继续下一章之前，先来看一下本章都学到了什么。

√ 算法。算法是对如何完成一项任务的详尽描述。实际上，在编写程序的时候，就是要使用计算机能够理解的语言(如Python)来描述算法。这类对机器友好的描述就叫做程序，程序主要包含表达式和语句。

√ 表达式。表达式是计算机程序的组成部分，它用于表示值。距离来说，2+2是表达式，表示数值4。简单的表达式就是通过使用运算符(如+或%)和函数(如pow)对字面值(比如2或者"Hello")进行处理而构建起来的。通过把简单的表达式联合起来可以建立更加复杂的表达式(如(2+2)*(3-1))。表达式也可以包含变量。

√ 变量。变量是一个名字，它表示某个值。通过x=2这样的赋值可以为变量赋予新的值。赋值也是一类语句。

√ 语句。语句是告诉计算机做某些事情的指令。它可能涉及改变变量(通过赋值)、向屏幕打印内容(如print "Hello, world!")、导入模块或者许多其他操作。

√ 函数。Python中的函数就像数学中的函数：它们可以带有参数，并且返回值(第六章会介绍如何编写自定义函数)。

√ 模块。模块是一些对Python功能的扩展，它可以被导入到Python中。例如，math模块提供了很多有用的数学函数。

√ 程序。本章之前的内容已经介绍过编写、保存和运行Python程序的实际操作了。

√ 字符串。字符串非常简单——就是文本片段，不过，还有很多与字符串相关的知识需要了解。在本章中，你已经看到很多种书写字符串的方法。第三章将会介绍更多字符串的使用方式。

1.12.1 本章的新函数

```
abs(number)                         返回数字的绝对值
cmath.sqrt(number)                  返回平方根，也可以应用于负数
float(object)                       将字符串和数字转换为浮点数
help()                              提供交互式帮助
input(prompt)                       获取用户输入
int(object)                         将字符串和数字转换为整数
long(object)                        将字符串和数字转换为长整型数
math.ceil(number)                   返回数的上入整数，返回值的类型为浮点数
math.floor(number)                  返回数的下入整数，返回值的类型为浮点数
math.sqrt(number)                   返回平方根，不适用于负数
pow(x, y[, z])                      返回x的y次方幂(所得结果对z取模)
raw_input(prompt)                   获取用户输入，结果被看做原始字符
repr(object)                        返回值的字符串表示形式
round(number[, ndigits)             根据给定的精度对数字进行四舍五入
str(object)                         将值转换为字符串
```

1.12.2 接下来学什么

表达式的基础知识已经讲解完毕，接下来要探讨更高级一点的内容：数据结构。你将学习到如何不再直接和简单的值(如数字)打交道，而是把它们集中起来处理，存储在更加复杂的结构中，如列表(list)和字典(dictionary)。另外，我们还将深入了解字符串。在第五章中，将会介绍更多关于语句的知识。之后，编写漂亮的程序就手到擒来了。

第二章 列表和元组

> 来源：http://www.cnblogs.com/Marlowes/p/5293195.html
>
> 作者：Marlowes

本章将引入一个新的概念：数据结构。数据结构是通过某种方式(例如对元素进行编号)组织在一起的数据元素的集合，这些数据元素可以是数字或者字符，甚至可以是其他数据结构。在Python中，最基本的数据结构是序列(sequence)，序列中的每个元素被分配一个序号——即元素的位置，也称为索引。第一个索引是`0`，第二个则是`1`，以此类推。

注：日常生活中，对某些东西计数或者编号的时候，可能会从`1`开始。所以*Python*使用的编号机制可能看起来很奇怪，但这种方法其实非常自然。在后面的章节中可以看到，这样做的一个原因是也可以从最后一个元素开始计数；序列中的最后一个元素标记为`-1`，倒数第二个元素为`-2`，以此类推。这就意味着我们可以从第一个元素向前或向后计数了，第一个元素位于最开始，索引为`0`.使用一段时间后，读者就会习惯于这种计数方式了。

本章首先对序列作一个概览，接下来讲解对所有序列(包括元组和列表)都通用的操作。这些操作也同样适用于字符串。尽管下一章才会全面介绍有关字符串操作的内容，但是本章的一些例子已经用到了字符串操作。

在完成了基本介绍后，会开始学习如何使用列表，同时看看它有什么特别之处。然后讨论元组。元组除了不能更改之外，其他的性质和列表都很类似。

2.1 序列概述

Python包含6中内建的序列，本章重点讨论最常用的两种类型：列表和元组。其他的内建序列类型有字符串(将在下一章再次讨论)、`Unicode`字符串、`buffer`对象和`xrange`对象。

列表和元组的主要却别在于：列表可以修改，元组则不能。也就是说如果要根据要求来添加元素，那么列表可能会更好用；而处于某些原因，序列不能修改的时候，使用元组则更为合适。使用后者通常是技术性的，它与Python内部的运作方式有关。这也是内建函数会返回元组的原因。一般来说，在自己编写的程序中，几乎在所有的情况下都可以用列表代替元组(第四章将会介绍一个需要注意的例外情况：使用元组作为字典的键。在这种情况下，因为键不可更改，所以就不能使用列表)。

在需要操作一组数值的时候，序列很好用。可以用序列表示数据库中一个人的信息——第1个元素是姓名，第2个元素是年龄。根据上述内容编写一个列表(列表的各个元素通过逗号分隔，写在方括号中)，如下所示：

```
>>> info = ["XuHoo", 19]
```

同时，序列也可以包含其他的序列，因此，构建如下的一个人员信息的列表也是可以的，这个列表就是你的数据库：

```
>>> user_1 = ["XuHoo", 19] >>> user_2 = ["Marlowes", 19] >>> database = [user_1, user_
2] >>> database
[['XuHoo', 19], ['Marlowes', 19]]
```

注：*Python*之中还有一种名为容器*(container)*的数据结构。容器基本上是包含到其他对象的任意对象。序列*(*例如列表和元组*)*和映射*(*例如字典*)*是两类主要的容器。序列中的每个元素都有自己的编号，而映射中的每个元素则有一个名字*(*也称为键*)*。在第四章会介绍更多有关映射的知识。至于既不是序列也不是映射的容器类型，集合*(* `set` *)*就是一个例子，请参见第十章的相关内容。

2.2 通用序列操作

所有序列类型都可以进行某些特定的操作。这些操作包括：索引*(indexing)*、分片*(slicing)*、加*(adding)*、乘*(multiplying)*以及检查某个元素是否属于序列的成员*(*成员资格*)*。除此之外，Python还有计算序列长度、找出最大元素和最小元素的内建函数。

注：本节有一个重要的操作没有提到——迭代*(iteration)*。对序列进行迭代的意思是：依次对序列中的每个元素重复执行某些操作。更多信息请参见*5.5*节。

2.2.1 索引

序列中的所有元素都是有编号的——从 `0` 开始递增。这些元素可以通过编号分别访问，如下例所示：

```
>>> greeting = "Hello"
>>> greeting[0] 'H'
```

注：字符串就是一个由字符组成的序列。索引 `0` 指向第*1*个元素，在这个例子中就是字母 `H` 。

这就是索引。可以通过索引获取元素。所有序列都可以通过这种方式进行索引。使用负数索引时，Python会从右边，也就是从最后1个元素开始计数。最后1个元素的位置编号是 `-1` (不是 `-0` ，因为那会和第1个元素重合)：

```
>>> greeting[-1] 'o'
```

字符串字面值(就此而言，其他序列字面量亦可)能够直接使用索引，而不需要一个变量引用它们。两种做法的效果是一样的：

```
>>> "Hello"[-1]  # String
'o'
>>> ["H", "e", "l", "l", "o"][-1]  # List
'o'
```

如果一个函数调用返回一个序列，那么可以直接对返回结果进行索引操作。例如，假设你只对用户输入年份的第四个数字感兴趣，那么，可以进行如下操作：

```
>>> fourth = raw_input("Year: ")[3]
Year: 1997
>>> fourth '7'
```

代码清单2-1是一个示例程序，它要求输入年、月(1~12的数字)、日(1~31)，然后打印出相应日期的月份名称，等等。

```
 1 #!/usr/bin/env python
 2 # coding=utf-8
 3
 4 # 根据给定的年月日，以数字形式打印出日期
 5 months = [ 6     "January",
 7     "February",
 8     "March",
 9     "April",
10     "May",
11     "June",
12     "July",
13     "August",
14     "September",
15     "October",
16     "November",
17     "December"
18 ]
19
20 # 以1~31的数字作为结尾的列表
21 endings = ["st", "nd", "rd"] + 17 * ["th"] \
22         + ["st", "nd", "rd"] + 7 * ["th"] \
23         + ["st"]
24
25 year = raw_input("Year: ")
26 month = raw_input("Month(1~12): ")
27 day = raw_input("Day(1~31): ")
28
29 month_number = int(month) 30 day_number = int(day)
31
32 # 记得要将月份和天数减1，以获得正确的索引
33 month_name = months[month_number - 1]
34 ordinal = day + endings[day_number - 1]
35
36 print month_name + " " + ordinal + ", " + year
```

Code_Listing 2-1

以下是程序执行的一部分结果：

```
Year: 1997 Month(1~12): 9 Day(1~31): 10 September 10th, 1997
```

2.2.2 分片

与使用索引来访问单个元素类似，可以使用分片操作来访问一定范围内的元素。分片通过冒号隔开的两个索引来实现：

```
>>> tag = '<a href="http://www.python.org">Python web site</a>'
>>> tag[9:30] 'http://www.python.org'
>>> tag[32:-4] 'Python web site'
```

分片操作对于提取序列的一部分是很用的。而编号在这里显得尤为重要。第1个索引是要提取的第1个元素的编号，而最后的索引则是分片之后剩余部分的第1个元素的编号。

```
>>> numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] >>> numbers[3:6]
[4, 5, 6] >>> numbers[0:1]
[1]
```

简而言之，分片操作的实现需要提供两个索引作为边界，第1个索引的元素是包含在分片内的，而第2个则不包含在分片内。

1. 优雅的捷径

假设需要访问最后3个元素(根据先前的例子)，那么当然可以进行显示的操作：

```
>>> numbers[7:10]
[8, 9, 10]
```

现在，索引 `10` 指向的是第11个元素——这个元素并不存在，却是在最后一个元素之后(为了让分片部分能够包含列表的最后一个元素，必须提供最后一个元素的下一个元素所对应的索引作为边界)。明白了吗？

现在，这样的做法是可行的。但是，如果需要从列表的结尾开始计数呢？

```
>>> numbers[-3:-1]
[8, 9]
```

看来并不能以这种方式访问最后的元素。那么使用索引 `0` 作为最后一步的下一步操作所使用的元素，结果又会怎么样呢？

```
>>> numbers[-3:0]
[]
```

这并不是我们所要的结果。实际上，只要分片中最左边的索引比它右边的晚出现在序列中(在这个例子中是倒数第3个比第1个晚出现)，结果就是一个空的序列。幸好，可以使用一个捷径：如果分片所得部分包括序列结尾的元素，那么，只需置空最后一个索引即可。

```
>>> numbers[-3:]
[8, 9, 10]
```

这种方法同样适用于序列开始的元素：

```
>>> numbers[:3]
[1, 2, 3]
```

实际上，如果需要复制整个序列，可以将两个索引都置空：

```
>>> numbers[:]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

代码清单2-2是一个小程序，它会提示输入URL(假设它的形式为 http://www.somedomainname.com)，然后提取域名。

```
1 #!/usr/bin/env python
2 # coding=utf-8
3
4 # 对http://www.something.com形式的URL进行分割
5
6 url = raw_input("Please enter the URL: ")
7 domain = url[11:-4]
8
9 print "Domain name: " + domain
```

Code_Listing 2-2

以下是程序运行的示例：

```
Please enter the URL: http://www.python.org

Domain name: python
```

2.更大的步长

进行分片的时候，分片的开始和结束点需要进行指定(不管是直接还是间接)。而另外一个参数(在Python2.3加入到内建类型)——步长(step length)——通常都是隐式设置的。在普通的分片中，步长是 `1` ——分片操作就是按照这个步长逐个遍历序列的元素，然后返回开始和结束点之间的所有元素。

```
>>> numbers[0:10:1]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

在这个例子中，分片包含了另外一个数字。没错，这就是步长的显示设置。如果步长被设置为比1大的数，那么就会跳过某些元素。例如，步长设为2的分片包括的是从头开始到结束每隔1个的元素。

```
>>> numbers[0:10:2]
[1, 3, 5, 7, 9] >>> numbers[3:6:3]
[4]
```

之前提及的捷径也可以使用。如果需要将每4个元素中的第1个提取出来，那么只要将步长设置为 `4` 即可：

```
>>> numbers[::4]
[1, 5, 9]
```

当然，步长不能为 `0` (那不会执行)，但步长可以是负数，此时分片从右到左提取元素：

```
>>> numbers[8:3:-1]
[9, 8, 7, 6, 5]
>>> numbers[10:0:-2]
[10, 8, 6, 4, 2]
>>> numbers[0:10:-2]
[]
>>> numbers[::-2]
[10, 8, 6, 4, 2]
>>> numbers[5::-2]
[6, 4, 2]
>>> numbers[:5:-2]
[10, 8]
```

在这里要得到正确的分片结果需要动些脑筋。开始点的元素(最左边的元素)包括在结果之中，而结束点的元素(最右边的元素)则不在分片之内。当使用一个负数作为步长时，必须让开始点(开始索引)大于结束点。在没有明确指定开始点和结束点的时候，正负数的使用可能会带来一些混淆。不过在这种情况下Python会进行正确的操作：对于一个正数步长，Python会从序列的头部开始向右提取元素，直到最后一个元素；而对于负数步长，则是从序列的尾部开始向左提取元素，直到第一个元素。

2.2.3 序列相加

通过使用加运算符可以进行序列的连接操作：

```
>>> [1, 2, 3] + [4, 5, 6]
[1, 2, 3, 4, 5, 6] >>> "Hello, " + "world!"
'Hello, world!'
>>> [1, 2, 3] + "wrold!" Traceback (most recent call last):
  File "<stdin>", line 1, in <module> TypeError: can only concatenate list (not "str")
 to list
```

正如错误信息所提示的，列表和字符串是无法连接在一起的，尽管它们都是序列。简单来说，两种相同类型的序列才能进行连接操作。

2.2.4 乘法

用数字 `x` 乘以一个序列会生成新的序列，而在新的序列中，原来的序列将被重复 `x` 次。

```
>>> "Python" * 5
'PythonPythonPythonPythonPython'
>>> [19] * 10 [19, 19, 19, 19, 19, 19, 19, 19, 19, 19]
```

None、空列表和初始化

空列表可以简单地通过两个中括号进行表示(`[]`)——里面什么东西都没有。但是，如果想创建一个占用十个元素空间，却不包括任何有用内容的列表，又该怎么办呢？可以像前面那样使用 `[19]*10` ，或者使用 `[0]*10` ，这会更加实际一些。这样就生成了一个包括10个0的列表。然而，有时候可能会需要一个值来代表空值——意味着没有在里面放置任何元素。这个时候就需要使用 `None` 。 `None` 是一个Python的内建值，它的确切含义是“这里什么也没有”。因此，如果想初始化一个长度为 `10` 的列表，可以按照下面的例子来实现：

```
>>> sequence = [None] * 10
>>> sequence
[None, None, None, None, None, None, None, None, None, None]
```

代码清单2-3的程序会在屏幕上打印一个由字符组成的“盒子”，而这个“盒子”在屏幕上居中而且能根据用户输入的句子自动调整大小。

代码可能看起来很复杂，但只使用基本的算法——计算出有多少个空格、破折号等字符，然后将它们放置到合适的位置即可。

```
 1 #!/usr/bin/env python
 2 # coding=utf-8
 3
 4 # 以正确的宽度在居中的“盒子”内打印一个句子
 5
 6 # 注意，整数除法运算符(//)只能用在Python2.2以及后续的版本，在之前的版本中，只能使用普通除法(/)
 7
 8 sentence = raw_input("Sentence: ")
 9
10 screen_width = 80
11 text_width = len(sentence) 12 box_width = text_width + 6
13 left_margin = (screen_width - box_width) // 2
14
15 print
16 print " " * left_margin + "+" + "-" * (box_width - 2) + "+"
17 print " " * left_margin + "| " + " " * text_width + " |"
18 print " " * left_margin + "| " + sentence + " |"
19 print " " * left_margin + "| " + " " * text_width + " |"
20 print " " * left_margin + "+" + "-" * (box_width - 2) + "+"
21 print
```

Code_Listing 2-3

下面是该例子的运行情况：

```
Sentence: He's a very naughty boy!

                         +----------------------------+
                          |                          |
                          | He's a very naughty boy! |
                          |                          |
                         +----------------------------+
```

2.2.5 成员资格

为了检查一个值是否在序列中，可以使用 `in` 运算符。该运算符和之前已经讨论过的(例如 `+` 、 `*` 运算符)有一点不同。这个运算符检查某个条件是否为真，然后返回相应的值：条件为真返回 `True` ，条件为假返回 `False` 。这样的运算符叫做布尔运算符，而返回的值叫做布尔值。第五章的条件语句部分会介绍更多关于布尔表达式的内容。

以下是一些使用了in运算符的例子：

```
>>> permissions = "rw"
>>> "w" in permissions
True >>> "x" in permissions
False >>> users = ["mlh", "foo", "bar"] >>> raw_input("Please enter your user name: ")
 in users  # 用户名存在
Please enter your user name: mlh
True >>> raw_input("Please enter your user name: ") in users  # 用户名不存在
Please enter your user name: Marlowes
False >>> subject = "$$$ Get rich now!!! $$$"
>>> "$$$" in subject
True
```

最初的两个例子使用了成员资格测试分别来检查 `"w"` 和 `"x"` 是否出现在字符串 `permissions` 中。在UNIX系统中，这两行代码可以作为查看文件可写和可执行权限的脚本。接下来的例子则是检查所提供的用户名是否存在用户列表中。如果程序需要执行某些安全策略，那么这个检查就派上用场了(在这种情况下，可能还需要使用密码)。最后一个例子可以作为垃圾邮件过滤器的一部分，它可以检查字符串 `subject` 是否包含字符串 `"$$$"` 。

注：最后一个检查字符串是否包含 *`"$$$"`* 的例子有些不同。一般来说， *`in`* 运算符会检查每一个对象是否为某个序列*(*或者是其他数据集合*)*的成员*(*也就是元素*)*。然后，字符串唯一的成员或者元素就是它的字符。下面的例子就说明了这一点：

```
>>> "P" in "Python" True
```

实际上，在早期的*Python*版本中，以上代码是唯一能用于字符串成员资格检查的方法——也就是检查某个字符是否存在于一个字符串中。如果尝试去检查更长的子字符串*(*例如*"$$$")*，那么会得到一个错误信息*(*这个操作会引发*TypeError*，即类型错误*)*。为了实现这个功能，我们必须使用相关的字符串方法。第三章会介绍更多相关的内容。但是从*Python2.3*起， *`in`* 运算符也能实现这个功能了。

代码清单2-4给出了一个查看用户输入的用户名和PIN码是否存在于数据库(实际上是一个列表)中的程序。如果用户名/PIN码这一数值对存在于数据库中，那么就在屏幕上打印 `"Access granted"` (第一章已经提到过 `if` 语句，第五章还将对其进行全面讲解)。

```
 1 #!/usr/bin/env python
 2
 3 # 检查用户名和PIN码
 4
 5 database = [
 6     ["albert", "123"],
 7     ["dilbert", "3521"],
 8     ["smith", "6542"],
 9     ["jones", "5634"]
10 ]
11
12 username = raw_input("Please enter your username: ")
13 pin = raw_input("Please enter your PIN: ")
14
15 if [username, pin] in database: 16     print "Access granted"
```

Code_Listing 2-4

2.2.6 长度、最小值和最大值

内建函数`len`、`min`和`max`非常有用。`len`函数返回序列中所包含元素的数量，`min`函数和`max`函数则分别返回序列中最大和最小元素(在第五章的“比较运算符”部分会更加详细介绍对象比较的内容)。

```
>>> numbers = [100, 34, 678] >>> len(numbers) 3
>>> max(numbers) 678
>>> min(numbers) 34
>>> max(2, 3) 3
>>> min(9, 3, 2, 5) 2
```

根据上述解释，我们可以很容易地理解例子中的各个操作是如何实现的，除了最后两个表达式可能会让人有些迷惑。在这里，`max`函数和`min`函数的参数并不是一个序列，而是以多个数字直接作为参数。

2.3 列表：Python的“苦力”

在前面的例子中已经用了很多次列表，它的强大之处不言而喻。本节会讨论列表不同于元组和字符串的地方：列表是可变的——可以改变列表的内容，并且列表有很多有用的、专门的方法。

2.3.1 list函数

因为字符串不能像列表一样被修改，所以有时根据字符串创建列表会很有用。`list`函数(它实际上是一种类型而不是函数，但在这里两者的区别并不重要)可以实现这个操作：

```
>>> list("Hello")
['H', 'e', 'l', 'l', 'o']
```

注意，`list`函数适用于所有类型的序列，而不只是字符串。

注：可以用下面的表达式将一个由字符(如前面代码中的)组成的列表转换为字符串：

```
''.join(somelist)
```

在这里，*somelist*是需要转换的列表。要了解这行代码真正的含义，请参考第三章有关*join*函数的部分。

2.3.2 基本的列表操作

列表可以使用所有适用于序列的标准操作，例如索引、分片、连接和乘法。有趣的是，列表是可以修改的。本节会介绍一些可以改变列表的方法：元素赋值、元素删除、分片赋值以及列表方法(请注意，并不是所有的列表方法都能真正地改变列表)。

1.改变列表：元素赋值

改变列表是很容易的，只需要使用第一章提到的普通赋值语句即可。然而，我们并不会使用`x=2`这样的语句进行赋值，而是使用索引标记来为某个特定的、位置明确的元素赋值。如`x[1]=2`。

```
>>> x = [1, 1, 1] >>> x[1] = 2
>>> x
[1, 2, 1]
```

注：不能为一个位置不存在的元素进行赋值。如果列表的长度为*2*，那么不能为索引为*100*的元素进行赋值。如果要那样做，就必须创建一个长度为*101*(或者更长)的列表。请参考本章"*None*、空列表和初始化"一节。

2.删除元素

从列表中删除元素也很容易：使用`del`语句实现。

```
>>> names = ["Alice", "Beth", "Cecil", "Dee-Dee", "Earl"] >>> del names[2] >>> names
['Alice', 'Beth', 'Dee-Dee', 'Earl']
```

注意`Cecil`是如何彻底消失的，并且列表的长度也从`5`变为了`4`。除了删除列表中的元素，`del`语句还能用于删除其他元素。它可以用于字典元素(请参考第四章)甚至是其他变量得删除操作，有关这方面的详细介绍，请参见第五章。

3.分片赋值

分片是一个非常强大的特性，分片赋值操作则更加显现它的强大。

```
>>> name = list("Perl") >>> name
['P', 'e', 'r', 'l'] >>> name[2:] = list("ar") >>> name
['P', 'e', 'a', 'r']
```

程序可以一次为多个元素赋值了。可能有的读者会想：这有什么大不了的，难道就不能一次一个地赋吗？当然可以，但是在使用分片赋值时，可以使用与原序列不等长的序列将分片替换：

```
>>> name = list("Perl") >>> name[1:] = list("ython") >>> name
['P', 'y', 't', 'h', 'o', 'n']
```

分片赋值语句可以在不需要替换任何原有元素的情况下插入新的元素：

```
>>> numbers = [1, 5] >>> numbers[1:1] = [2, 3, 4] >>> numbers
[1, 2, 3, 4, 5]
```

这个程序只是“替换”了一个空的分片，因此实际的操作是插入了一个序列。以此类推，通过分片赋值来删除元素也是可行的。

```
>>> numbers
[1, 2, 3, 4, 5] >>> numbers[1:4] = [] >>> numbers
[1, 5]
```

上面的例子结果和 `del numbers[1:4]` 的一样。接下来请读者自己尝试利用1之外的步长，甚至是负数进行分片吧。

2.3.3 列表方法

之前的章节中已经介绍了什么是函数，那么现在来看看另外一个与函数密切相关的概念——方法。

方法是一个与某些对象有紧密联系的函数，对象可能是列表、数字，也可能是字符串或者其他类型的对象。一般来说，方法可以这样进行调用：

对象.方法(参数)

除了对象被放置到方法名之前，并且两者之间用一个点号隔开，方法调用与函数调用很类似。第七章将对方法到底是什么进行更详细的解释。列表提供了几个方法，用于检查或者修改其中的内容。

1. `append`

`append` 方法用于在列表末尾追加新的对象：

```
>>> lst = [1, 2, 3] >>> lst.append(4) >>> lst
[1, 2, 3, 4]
```

为什么我选择了如此糟糕的变量名 `lst` ，而不是使用 `list` 来表示一个列表呢？原因在于 `list` 是一个内建函数(实际上，从Python2.2开始， `list` 就是一个类型而不是函数了。(`tuple` 和 `str` 也是如此)如果想了解完整的说明，请参见9.3.2节)。如果使用 `list` 作为变量名，我就无法调用 `list` 函数了。根据给定的应用程序可以定义更好的变量名，像 `lst` 这样的变量名是毫无意义的。所以，如果需要定义一个价格列表，那么就应该使用 `prices` 、 `prices_of_eggs` ，或者 `pricesOfEggs` 作为变量名。

注意，下面的内容很重要： `append` 方法和其他一些方法类似，只是在恰当位置修改原来的列表。这意味着，它不是简单地返回一个修改过的新列表——而是直接修改原来的列表。一般来说这正是你想要的，但是在某些情况下，这样也会带来其他麻烦。在本章稍后讲述 `sort` 方法时，我将再次讨论这个问题。

2. `count`

`count` 方法统计某个元素在列表中出现的次数：

```
>>> ["to", "be", "or", "not", "to", "be"].count("to") 2
>>> x = [[1, 2], 1, 1, [2, 1, [1, 2]]] >>> x.count(1) 2
>>> x.count([1, 2]) 1
```

3. `extend`

`extend` 方法可以在列表的末尾一次性追加另一个序列中的多个值。换句话说，可以用新的列表扩展原有的列表：

```
>>> a = [1, 2, 3] >>> b = [4, 5, 6] >>> a.extend(b) >>> a
[1, 2, 3, 4, 5, 6]
```

这个操作看起来很像连接操作，两者最主要的区别在于： `extend` 方法修改了被扩展的序列(在这个例子中，就是 `a`)。而原始的连接操作则不然，它会返回一个全新的列表：

```
>>> a = [1, 2, 3] >>> b = [4, 5, 6] >>> a + b
[1, 2, 3, 4, 5, 6] >>> a
[1, 2, 3]
```

你可以看到连接的列表与之前例子中被扩展的列表是一样的，但是这一次它并没有被修改。这是因为原始的连接操作创建了一个包含 `a` 和 `b` 副本的新列表。如果需要如下例所示的操作，那么连接操作的效率会比 `extend` 方法低。

```
>>> a = a + b
```

同样，这里也不是一个原位置操作，它并不会修改原来的列表。

我们可以使用分片赋值来实现相同的结果：

```
>>> a = [1, 2, 3] >>> b = [4, 5, 6] >>> a[len(a):] = b >>> a
[1, 2, 3, 4, 5, 6]
```

虽然这么做是可行的，但是代码的可读性就不如使用 `extend` 方法了。

4. `index`

`index` 方法用于从列表中找出某个值第一个匹配项的索引位置：

```
>>> knights = ["We", "are", "the", "knights", "who", "say", "ni"] >>> knights.index("w
ho") 4
>>> knights.index("herring")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module> ValueError: 'herring' is not in list
```

当搜索单词 `who` 的时候，就会发现它在索引号为 `4` 的位置。然而，当搜索 `"herring"` 的时候，就会引发一个异常，因为这个单词没有被找到。

5. `insert`

`insert` 方法用于将对象插入列表中：

```
>>> numbers = [1, 2, 3, 5, 6, 7] >>> numbers.insert(3, "four") >>> numbers
[1, 2, 3, 'four', 5, 6, 7]
```

与 `extend` 方法一样， `insert` 方法的操作也可以用分片赋值来实现。

```
>>> numbers = [1, 2, 3, 5, 6, 7] >>> numbers[3:3] = ["four"] >>> numbers
[1, 2, 3, 'four', 5, 6, 7]
```

这样做有点新奇，但是它的可读性绝对不如 `insert` 方法。

6. `pop`

`pop` 方法会移除列表中的一个元素(默认是最后一个)，并且返回该元素的值：

```
>>> x = [1, 2, 3] >>> x.pop() 3
>>> x
[1, 2] >>> x.pop(0) 1
>>> x
[2]
```

注： *`pop`* *方法是唯一一个既能修改列表又返回元素值(除了* *`None`* *)的列表方法。*

使用 `pop` 方法可以实现一种常见的数据结构——栈。栈的原理就像堆放盘子那样。只能在顶部放盘子，同样，也只能从顶部拿走一个盘子。最后被放入栈堆的最先被移除(这个原则成为LIFO，即后进先出)。

对于上述的两个栈操作(放入和移出)，它们有大家都认可的称谓——入栈(`push`)和出栈(`pop`)。Python没有入栈方法，但可以使用`append`方法来代替。`pop`方法和`append`方法的操作结果恰好相反，如果入栈(或者追加)刚刚出栈的值，最后得到的结果还是原来的栈。

```
>>> x = [1, 2, 3] >>> x.append(x.pop()) >>> x
[1, 2, 3]
```

注：如果需要实现一个先进先出(FIFO)的队列(`queue`)，那么可以使用`insert(0, ...)`来代替`append`方法。或者，也可以继续使用`append`方法，但必须用`pop(0)`来代替`pop()`。更好的解决方案是使用`collection`模块中的`deque`对象。要了解更详细的信息，请参见第十章

7. `remove`

`remove`方法用于移除列表中某个值的第一个匹配项：

```
>>> x = ["to", "be", "or", "not", "to", "be"] >>> x.remove("be") >>> x
['to', 'or', 'not', 'to', 'be'] >>> x.remove("bee")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module> ValueError: list.remove(x): x not in list
```

可以看到：只有第一次出现的值被移除了，而不存在于列表中的值(比如例子中的`"bee"`)是不会移除的。

值得注意的是，`remove`是一个没有返回值的原位置改变的方法。它修改了列表却没有返回值，这与`pop`方法相反。

8. `reverse`

`reverse`方法将列表中的元素反向存放(我猜你们对此不会特别惊讶)：

```
>>> x = [1, 2, 3] >>> x.reverse() >>> x
[3, 2, 1]
```

请注意，该方法也改变了列表但不反悔值(就像`remove`和`sort`)。

注：如果需要对一个序列进行反向迭代，那么可以使用`reversed`函数。这个函数并不返回一个列表，而是返回一个迭代器(`iterator`)对象(第九章介绍了更多关于迭代器的内容)。尽管如此，使用`list`函数把返回值的对象转换成列表也是可行的：

```
>>> x = [4, 6, 2, 1, 7, 9] >>> list(reversed(x))
[9, 7, 1, 2, 6, 4]
```

9.sort

`sort` 方法用于在原位置(从Python2.3开始，`sort` 方法使用了固定的排序算法)对列表进行排序。在“原位置排序”意味着改变原来的列表，从而让其中的元素未能按一定的顺序排列，而不是简单地返回一个已排序的列表副本。

```
>>> x = [4, 6, 2, 1, 7, 9] >>> x.sort() >>> x
[1, 2, 4, 6, 7, 9]
```

前面介绍过了几个改变列表却不返回值的方法，在大多数情况下这样的行为方式是很合常理的(例如 `append` 方法)。但是，`sort` 方法的这种行为方式需要重点讲解一下，因为很多人都被 `sort` 方法弄糊涂了。当用户需要一个排好序的列表副本，同时又保留原有列表不变的时候，问题就出现了。为了实现这个功能，我们自然而然就想到了如下的做法(实际是错误的)：

```
>>> x = [4, 6, 2, 1, 7, 9] >>> y = x.sort()  # Don't do this!
>>> print y
None
```

因为 `sort` 方法修改了x却返回了空值，那么最后得到的是已排序的 `x` 以及值为 `None` 的 `y` 。实现这个功能的正确方法是，首先把 `x` 的副本赋值给 `y` ，然后对 `y` 进行排序，如下例所示：

```
>>> x = [4, 6, 2, 1, 7, 9] >>> y = x[:] >>> y.sort() >>> x
[4, 6, 2, 1, 7, 9] >>> y
[1, 2, 4, 6, 7, 9]
```

再次调用 `x[:]` 得到的是包含了x所有元素的分片，这是一种很有效率的赋值整个列表的方法。只是简单地把 `x` 赋值给 `y` 是没有用的，因为这样做就让 `x` 和 `y` 都指向同一个列表了。

```
>>> x = [4, 6, 2, 1, 7, 9] >>> y = x >>> y.sort() >>> x
[1, 2, 4, 6, 7, 9] >>> y
[1, 2, 4, 6, 7, 9]
```

另一种获取已排序的列表副本的方法是，使用 `sorted` 函数：

```
>>> x = [4, 6, 2, 1, 7, 9] >>> y = sorted(x) >>> x
[4, 6, 2, 1, 7, 9] >>> y
[1, 2, 4, 6, 7, 9]
```

这个函数实际上可以用于任何序列，却总是返回一个列表(`sorted` 函数可以用于任何可迭代的对象。有关可迭代对象的详细内容，请参见第九章)：

```
>>> sorted("Python")
['P', 'h', 'n', 'o', 't', 'y']
```

如果想把一些元素按相反的顺序排序，可以先使用 `sort` (或者 `sorted`)，然后再调用 `reverse` 方法(注意，需要分两次对列表调用 `sort` 方法以及 `reverse` 方法。如果打算通过 `x.sort().reverse()` 来实现，会发现行不通，因为 `x.sort()` 返回的是 `None` 。当然， `sorted(x).reverse()` 是正确的做法)，或者也可以使用 `reverse` 参数，下一节将对此进行描述。

10.高级排序

如果希望元素能按照特定的方式进行排序(而不是 `sort` 函数默认的方式，即根据Python的默认排序规则按升序排列元素，第五章内对此进行详解)，那么可以通过 `compare(x, y)` 的形式自定义比较函数。 `compare(x, y)` 函数会在 `x < y` 时返回负数，在 `x > y` 时返回正数，如果 `x = y` 则返回0(根据你的定义)。定义好该函数之后，就可以提供给 `sort` 方法作为参数了。内建函数 `cmp` 提供了比较函数的默认实现方式：

```
>>> cmp(42, 32) 1
>>> cmp(99, 100) -1
>>> cmp(19, 19)
0

>>> numbers = [5, 2, 9, 7]

>>> numbers.sort(cmp)

>>> numbers

[2, 5, 7, 9]
```

`sort` 方法有另外两个可选的参数—— `key` 和 `reverse` 。如果要使用它们，那么就要通过名字来指定(这叫做关键字参数，请参见第六章以了解更多的内容)。参数 `key` 与参数 `cmp` 类似——必须提供一个在排序过程中使用的函数。然而，该函数并不是直接用来确定对象的大小，而是为每个元素创建一个键，然后所有元素根据键来排序。因此，如果要根据元素的长度进行排序，那么可以使用 `len` 作为键函数：

```
>>> x = ["aardvark", "abalone", "acme", "add", "aerate"] >>> x.sort(key=len) >>> x
['add', 'acme', 'aerate', 'abalone', 'aardvark']
```

另一个关键字参数 `reverse` 是简单的布尔值(`True` 或者是 `False` 。第五章会讲述更详细的内容)，用来指明列表是否要进行反向排序。

```
>>> x = [4, 6, 2, 1, 7, 9] >>> x.sort(reverse=True)  # True为反向排序
>>> x
[9, 7, 6, 4, 2, 1] >>> x = [4, 6, 2, 1, 7, 9]
>>> x.sort(reverse=False)  # Flase为正向排序
>>> x
[1, 2, 4, 6, 7, 9]
```

`cmp` 、 `key` 、 `reverse` 参数都可以用于 `sorted` 函数。在多数情况下，为 `cmp` 或 `key` 提供自定义函数是非常有用的。第六章将会讲述如何定义自己的函数。

注：如果想了解更多有关于排序的内容，可以查看Andrew Dalke的“Sorting Mini-HOWTO”，链接是：http://wiki.python.org/moin/HowTo/Sorting 。

2.4 元组：不可变序列

元组与列表一样，也是一种序列。唯一的不同是元组不能修改(元组和列表在技术实现上有一些不同，但是在实际使用时，可能不会注意到。而且，元组没有像列表一样的方法。)。(你可能注意到了，字符串也是如此)创建元组的语法很简单：如果你用逗号分隔了一些值，那么你就自动创建了元组。

```
>>> 1, 2, 3 (1, 2, 3)
```

元组也是(大部分时候是)通过圆括号括起来的：

```
>>> (1, 2, 3)
(1, 2, 3)
```

空元组可以用没有包含内容的两个圆括号来表示：

```
>>> ()
()
```

那么如何实现包括一个值的元组呢。实现方法有些奇特——必须加个逗号，即使只有一个值：

```
>>> 42
42
>>> 42,
(42,) >>> (42,)
(42,)
```

最后两个例子生成了一个长度为1的元组，而第一个例子根本不是元组。逗号是很重要的，只添加圆括号也是没用的：`(42)` 和 `42` 是完全一样的。但是，一个逗号却能彻底地改变表达式的值：

```
>>> 3 * (40 + 2) 126
>>> 3 * (40 + 2,)
(42, 42, 42)
```

2.4.1 tuple函数

`tuple` 函数的功能与 `list` 函数基本上是一样的：以一个序列作为参数并把它转换为元组(`tuple` 并不是真正的函数——而是一种类型。在之前讲述 `list` 函数的时候，我也提到了这一点。同时，与 `list` 函数一样，目前也可以放心地忽略这一点)。如果参数就是元组，那么该参数就会被原样返回：

```
>>> tuple([1, 2, 3])
(1, 2, 3) >>> tuple("abc")
('a', 'b', 'c') >>> tuple((1, 2, 3))
(1, 2, 3)
```

2.4.2 基本元组操作

元组其实并不复杂——除了创建元组和访问元组元素之外，也没有太多其他的操作，可以参照其他类型的序列来实现：

```
>>> x = 1, 2, 3
>>> x[1] 2
>>> x[0:2]
(1, 2)
```

元组的分片还是元组，就像列表的分片还是列表一样。

2.4.3 那么，意义何在

现在你可能会想到底有谁会需要像元组那样的不可变序列呢？难道就不能在不改变其中内容的时候坚持只用列表吗？一般来说这是可行的。但是由于以下两个重要的原因，元组是不可替代的。

√ 元组可以在映射(和集合的成员)中当做键使用——而列表则不行(本章导言部分提到过映射，更多有关映射的内容，请参看第四章)。

√ 元组作为很多内建函数和方法的返回值存在，也就是说你必须对元组进行处理。只要不尝试修改元组，那么，“处理”元组在绝大多数情况下就是把它们当做列表来进行操作(除非需要使用一些元组没有的方法，例如 `index` 和 `count`)。

一般来说，列表可能更能满足对序列的所有需求。

2.5 小结

让我们回顾本章所涵盖的一些最重要的内容。

√ 序列。序列是一种数据结构，它包含的元素都进行了编号(从 `0` 开始)。典型的序列包括列表、字符串和元组。其中，列表是可变的(可以进行修改)，而元组和字符串是不可变的(一旦创建了就是固定的)。通过分片操作可以访问序列的一部分，其中分片需要两个索引号来指出

分片的起始和结束位置。要想改变列表，则要对相应的位置进行赋值，或者使用赋值语句重写整个分片。

√ 成员资格。`in`操作符可以检查一个值是否存在于序列(或者其他的容器)中。对字符串使用`in`操作符是一个特例，它可以查找子字符串。

√ 方法。一些内建类型(例如列表和字符串，元组则不在其中)具有很多有用的方法。这些方法有些像函数，不过它们与特定值联系得更密切。方法是面向对象编程的一个重要的概念，稍后的第七章中会对其进行讨论。

2.5.1 本章的新函数

```
cmp(x, y)                         比较两个值
len(seq)                          返回序列的长度
list(seq)                         把序列转换成列表
max(args)                         返回序列或者参数集合中的最大值
min(args)                         返回序列或者参数集合中的最小值
reversed(seq)                     对序列进行反向迭代
sorted(seq)                       返回已排序的包含seq所有元素的列表
tuple(seq)                        把序列转换成元组
```

2.5.2 接下来学什么

序列已经介绍完了，下一章会继续介绍由字符组成的序列，即字符串。

第三章 使用字符串

> 来源：http://www.cnblogs.com/Marlowes/p/5312236.html
>
> 作者：Marlowes

读者已经知道了什么是字符串，也知道如何创建它们。利用索引和分片访问字符串中的单个字符也已经不在话下了。那么本章将会介绍如何使用字符串格式化其他的值(如打印特殊格式的字符串)，并简单了解一下利用字符串的分割、连接、搜索等方法能做些什么。

3.1 基本字符串操作

所有标准的序列操作(索引、分片、乘法、判断成员资格、求长度、取最小值和最大值)对字符串同样适用，上一章已经讲述了这些操作。但是，请记住字符串都是不可变的。因此，如下所示的项或分片赋值都是不合法的：

```
>>> website = "http://www.python.org"
>>> website[-3:] = "com" Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  TypeError: 'str' object does not support item assignment
```

3.2 字符串格式化：精简版

如果初次接触Python编程，那么Python提供的所有字符串格式化功能可能用不到太多。因此，这里只简单介绍一些主要的内容。如果读者对细节感兴趣，可以参见下一章，否则可以直接阅读3.4节。

字符串格式化使用字符串格式化操作符(这个名字还是很恰当的)即百分号 `%` 来实现。

注：`%` 也可以用作模运算*(*求余*)*操作符。

在 `%` 的左侧放置一个字符串(格式化字符串)，而右侧则放置希望被格式化的值。可以使用一个值，如一个字符串或者数字，也可以使用多个值的元组或者下一章将会讨论的字典(如果希望格式化多个值的话)，这部分内容将在下一章进行讨论。一般情况下使用元组：

```
>>> format = "Hello, %s. %s enough for ya?"
>>> values = ("world", "Hot")
>>> print format % values
Hello, world. Hot enough for ya?
```

注：如果使用列表或者其他序列代替元组，那么序列会被解释为一个值。只有元组和字典*(*将在第*4*章讨论*)*可以格式化一个以上的值。

格式化字符串的 `%s` 部分称为转换说明符(conversion specifier)，它们标记了需要插入转换值的位置。`s` 表示值会被格式化为字符串——如果不是字符串，则会用 `str` 将其转换为字符串。这个方法对大多数值都有效。其他转换说明符请参见本章后面的表3-1.

注：如果要在格式化字符串里面包括百分号，那么必须使用 `%%` ，这样*Python*就不会将百分号误认为是转换说明符了。

如果要格式化实数(浮点数)，可以使用 `f` 说明转换说明符的类型，同时提供所需要的精度：一个句点再加上希望保留的小数位数。因为格式化转换说明符总是以表示类型的字符结束，所以精度应该放在类型字符前面：

```
>>> format = "Pi with three decimals: %.3f"
>>> from math import pi
>>> print format % pi
Pi with three decimals: 3.142
```

模板字符串

string模块提供另外一种格式化值的方法：模板字符串。它的工作方式类似于很多UNIX Shell里的变量替换。如下所示，`substitute` 这个模板方法会用传递进来的关键字参数 `foo` 替换字符串中的 `$foo` (有关关键字参数的详细信息，请参看第六章)：

```
>>> from string import Template
>>> s = Template("$x, glorious $x!")
>>> s.substitute(x="slurm") 'slurm, glorious slurm!'
```

如果替换字段是单词的一部分，那么参数名就必须用括号括起来，从而准确指明结尾：

```
>>> s = Template("It's ${x}tastic!")
>>> s.substitute(x="slurm") "It's slurmtastic!"
```

可以使用 `$$` 插入美元符号：

```
>>> s = Template("Make $$ selling $x!")
>>> s.substitute(x="slurm") 'Make $ selling slurm!'
```

除了关键字参数之外，还可以使用字典变量提供值/名称对(参见第四章)。

```
>>> s = Template("A $thing must never $action.") >>> d = {}
>>> d["thing"] = "gentleman"
>>> d["action"] = "show his socks"
>>> s.substitute(d)
'A gentleman must never show his socks.'
```

方法 `safe_substitute` 不会因缺少值或者不正确使用 `$` 字符而出错(更多信息请参见Python库参考手册的4.1.2节)。

3.3 字符串格式化：完整版

格式化操作符的右操作数可以是任意类型，如果是元组或者映射类型(如字典)，那么字符串格式化将会有所不同。我们尚未涉及映射(如字典)，在此先了解一下元组。第四章还会详细介绍映射的格式化。

如果右操作数是元组的话，则其中的每一个元素都会被单独格式化，每个值都需要一个对应的转换说明符。

注：如果需要转换的元组作为转换表达式的一部分存在，那么必须将它用圆括号括起来，以避免出错。

```
>>> "%s plus %s equals %s" % (1, 1, 2)
'1 plus 1 equals 2'
# Lacks parentheses!
>>> "%s plus %s equals %s" % 1, 1, 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  TypeError: not enough arguments for format string
```

基本的转换说明符(与此相对应的是完整的转换说明符，也就是包括映射键的说明符，详细内容参见第四章)包括以下部分。注意，这些项的顺序是至关重要的。

(1) `%` 字符：标记转换说明符的开始。

(2)转换标志(可选)： `-` 表示左对齐； `+` 表示在转换值之前要加上正负号； `" "` (空白字符)表示整数之前保留空格； `0` 表示转换值若位数不够则用 `0` 填充。

(3)最小字段宽度(可选)：转换后的字符串至少应该具有该值指定的宽度。如果是 `*` ，则宽度会从值元组中读出。

(4)点(`.`)后跟精度值(可选)：如果转化的是实数，精度值就表示出现在小数点后的位数。如果转换的是字符串，那么该数字就表示最大字段宽度。如果是*，那么精度将会从元组中读出。

(5)转换类型：参见表3-1。

表3-1 字符串格式化转换类型

```
d，i          带符号的十进制整数
o             不带符号的八进制
u             不带符号的十进制
x             不带符号的十六进制(小写)
X             不带符号的十六进制(大写)
e             科学计数法表示的浮点数(小写)
E             科学计数法表示的浮点数(大写)
f，F          十进制浮点数
g             如果指数大于-4或者小于精度值则和e相同，其他情况与f相同
G             如果指数大于-4或者小于精度值则和E相同，其他情况与F相同
C             单字符(接受整数或者单字符字符串)
r             字符串(使用repr转换任意Python对象)
s             字符串(使用str转换任意Python对象)
```

接下来几个小节将对转换说明符的各个元素进行详细讨论。

3.3.1 简单转换

简单的转换只需要写出转换类型，使用起来很简单：

```
>>> "Price of eggs: $%d" % 42
'Price of eggs: $42'
>>> "Hexadecimal price of eggs: %x" % 42
'Hexadecimal price of eggs: 2a'
>>> from math import pi >>> "Pi: %f..." %pi
'Pi: 3.141593...'
>>> "Very inexact estimate of pi: %i" % pi
'Very inexact estimate of pi: 3'
>>> "Using str: %s" % 42L
'Using str: 42'
>>> "Using repr: %r" % 42L
'Using repr: 42L'
```

3.3.2 字段宽度和精度

转换说明符可以包括字段宽度和精度。字段宽度是转换后的值所保留的最小字符个数，精度(对于数字转换来说)则是结果中应该包含的小数位数，或者(对于字符串转换来说)是转换后的值所能包含的最大字符个数。

这两个参数都是整数(首先是字段宽度，然后是精度)，通过点号(`.`)分隔。虽然两个都是可选的参数，但如果只给出精度，就必须包含点号：

```
>>> "%10f" % pi  # 字段宽10
' 3.141593'
>>> "%10.2f" % pi  # 字段宽10，精度2
' 3.14'
>>> "%.2f" % pi  # 精度2
'3.14'
>>> "%.5s" % "Guido van Rossum"
'Guido'
```

可以使用 `*` (星号)作为字段宽度或者精度(或者两者都是用 `*`)，此时数值会从元组参数中读出：

```
>>> "%.*s" % (5, "Guido van Rossum")
'Guido'
```

3.3.3 符号、对齐和用0填充

在字段宽度和精度值之前还可以放置一个“标志”，该标志可以是零、加号、减号或空格。零表示数字将会用 `0` 进行填充。

```
>>> "%010.2f" % pi
'0000003.14'
```

注意，在`010`中开头的那个`0`并不意味着字段宽度说明符为八进制数，它只是个普通的Python数值。当使用`010`作为字段宽度说明符的时候，表示字段宽度为`10`，并且用`0`进行填充空位，而不是说字段宽度为`8`：

```
>>> 010
8
```

减号(`-`)用来左对齐数值：

```
>>> "%-10.2f" % pi
'3.14 '
```

可以看到，在数字的右侧多出了额外的空格。

而空白(`" "`)意味着在正数前加上空格。这在需要对齐正负数时会很有用：

```
>>> print ("%+5d" % 10) + "\n" + ("%+5d" % -10) +10
-10
```

代码清单3-1中的代码将使用星号字段宽度说明符来格式化一张包含水果价格的表格，表格的总宽度由用户输入。因为是由用户提供信息，所以就不能在转换说明符中将字段宽度硬编码。使用星号运算符就可以从转换元组中读出字段宽度。

```
 1 #!/usr/bin/env python
 2 # coding=utf-8
 3
 4 # 使用给定的宽度打印格式化后的价格列表
 5
 6 width = input("Please enter width: ")
 7
 8 price_width = 10
 9 item_width = width - price_width
10
11 header_format = "%-*s%*s"
12 format = "%-*s%*.2f"
13
14 print "=" * width 15
16 print header_format % (item_width, "Item", price_width, "Price")
17
18 print "-" * width 19
20 print format % (item_width, "Apples", price_width, 0.4)
21 print format % (item_width, "Pears", price_width, 0.5)
22 print format % (item_width, "Cantaloupes", price_width, 1.92)
23 print format % (item_width, "Dried Apricots (16 oz.)", price_width, 8) 24 print for
mat % (item_width, "Prunes (4 lbs.)", price_width, 12)
25
26 print "=" * width
```

Code_Listing 3-1

以下是程序运行示例：

```
Please enter width: 35
===================================
Item                          Price
-----------------------------------
Apples                         0.40
Pears                          0.50
Cantaloupes                    1.92
Dried Apricots (16 oz.)        8.00
Prunes (4 lbs.)               12.00
===================================
```

3.4 字符串方法

前面几节已经介绍了很多列表的方法，字符串的方法还要丰富得多，这是因为字符串从 `string` 模块中“继承”了很多方法，而在早期版本的Python中，这些方法都是作为函数出现的(如果真的需要的话，还是能找到这些函数的)。

因为字符串的方法是实在太多，在这里只介绍一些特别有用的。全部方法请参见附录B。在字符串的方法描述中，可以在本章找到关联到其他方法的参考(用“请参见”标记)，或请参见附录B。

但是字符串未死

尽管字符串方法完全来源于 `string` 模块，但是这个模块还包括一些不能作为字符串方法使用的常量和函数。 `maketrans` 函数就是其中之一，后面会将它和 `translate` 方法一起介绍。下面是一些有用的字符串常量(对于此模块的更多介绍，请参见Python库参考手册的4.1节)。

√ string.digits：包含数字0~9的字符串。

√ string.letters：包含所有字母(大写或小写)的字符串。

√ string.lowercase：包含所有小写字母的字符串。

√ string.printable：包含所有可打印字符的字符串。

√ string.punctuation：包含所有标点的字符串。

√ string.uppercase：包含所有大写字母的字符串。

字母字符串常量(例如 `string.letters`)与地区有关(也就是说，其具体值取决于Python所配置的语言)(在Python3.0中， `string.letters` 和其相关内容都会被移除。如果需要则应该使用 `string.ascii_letters` 常量代替)。如果可以确定自己使用的ASCII，那么可以在变量中使用 `ascii_` 前缀，例如 `string.ascii_letters` 。

3.4.1 `find`

`find` 方法可以在一个较长的字符串中查找子串。它返回子串所在位置的最左端索引。如果没有找到则返回 `-1` 。

```
>>> "With a moo-moo here, and a moo-moo there".find("moo") 7
>>> title = "Monty Python's Flying Circus"
>>> title.find("Monty")
0 >>> title.find("Python")  # 找到字符串
6
>>> title.find("Flying") 15
>>> title.find("Zirquss")  # 未找到字符串
-1
```

在第二章中我们初始了成员资格，我们在 `subject` 中使用了 `"$$$"` 表达式建立了一个垃圾邮件过滤器。也可以使用 `find` 方法(Python2.3以前的版本中也可用，但是 `in` 操作符只能用来查找字符串中的单个字符)：

```
>>> subject = "$$$ Get rich now!!! $$$"
>>> subject.find("$$$")
0
```

注：字符串的 *`find`* 方法并不返回布尔值。如果返回的是 *`0`* ，则证明在索引 *`0`* 位置找到了子串。

这个方法还可以接收可选的起始点和结束点参数：

```
>>> subject = "$$$ Get rich now!!! $$$"
>>> subject.find("$$$")
0 >>> subject.find("$$$", 1)  # 只提供起始点
20
>>> subject.find("!!!") 16
>>> subject.find("!!!", 0, 16)  # 提供起始点和结束点
-1
```

注意，由起始和终止值指定的范围(第二个和第三个参数)包含第一个索引，但不包含第二个索引。这在Python中是个惯例。

附录B： `rfind` 、 `index` 、 `rindex` 、 `count` 、 `startswith` 、 `endswith` 。

3.4.2 `join`

`join` 方法是非常重要的字符串方法，它是 `split` 方法的逆方法，用来连接序列中的元素：

```
>>> seq = [1, 2, 3, 4, 5]
>>> sep = "+"
>>> sep.join(seq)  # 连接数字列表
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  TypeError: sequence item 0: expected string, int found
>>> seq = ["1", "2", "3", "4", "5"]
>>> sep.join(seq)  # 连接字符串列表
'1+2+3+4+5'
>>> dirs = "", "usr", "bin", "env"
>>> "/".join(dirs)
'/usr/bin/env'
>>> print "C:" + "\\".join(dirs)
C:\usr\bin\env
```

可以看到，需要被连接的序列元素都必须是字符串。注意最后两个例子中使用了目录的列表，而在格式化时，根据UNIX和DOS/Windows的约定，使用了不同的分隔符号(在DOS版本中还增加了驱动器名)。

请参见：`split`。

3.4.3 `lower`

`lower`方法返回字符串的小写字母版。

```
>>> "Trondheim Hammer Dance".lower()
'trondheim hammer dance'
```

如果想要编写“不区分大小写”的代码的话，那么这个方法就派上用场了——代码会忽略大小写状态。例如，如果想在列表中查找一个用户名是否存在：列表包含字符串`"gumby"`，而用户输入的是`"Gumby"`，就找不到了：

```
>>> if "Gumby" in ["gumby", "smith", "jones"]:
        print "Found it!"
...
>>>
```

如果存储的是`"Gumby"`而用户输入`"gumby"`甚至是`"GUMBY"`，结果也是一样的。解决方法就是在存储和搜索时把所有名字都转换为小写。代码如下：

```
>>> name = "Gumby"
>>> names = ["gumby", "smith", "jones"]
>>> if name.lower() in names:
        print "Found it!"
...
Found it!
```

请参见：`translate`。

附录B：`islower`、`capitalize`、`swapcase`、`title`、`istitle`、`upper`、`isupper`。

标题转换

和`lower`方法相关的是`title`方法(参见附录B)，它会将字符串转换为标题——也就是所有单词的首字母大写，而其他字母小写。但是它使用的单词划分方法可能会得到并不自然的结果：

```
>>> "that's all folks".title()
"That'S All Folks"
```

再介绍另外一个`string`模块的`capwords`函数：

```
>>> import string
>>> string.capwords("that's all, folks")
"That's All, Folks"
```

当然，如果要得到正确首字母大写的标题(这要根据你的风格而定，可能要小写冠词、连词及5个字母以下的介词等)，那么还是得自己把握。

3.4.4 `replace`

`replace`方法返回某字符串的所有匹配项均被替换之后得到字符串。

```
>>> "This is a test".replace("is", "eez")
'Theez eez a test'
```

如果曾经用过文字处理程序中的“查找并替换”功能的话，就不会质疑这个方法的用处了。

请参见：`translate`。

附录B：`expandtabs`。

3.4.5 `split`

这是一个非常重要的字符串方法，它是`join`的逆方法，用来将字符串分隔成序列。

```
>>> "1+2+3+4+5".split("+")
['1', '2', '3', '4', '5']
>>> "/usr/bin/env".split("/")
['', 'usr', 'bin', 'env']
>>> "Using the default".split()
['Using', 'the', 'default']
```

注意，如果不提供任何分隔符，程序会把所有空格作为分隔符(空格、制表、换行等)。

请参见：`join`。

附录B：`rsplit`、`splitlines`。

3.4.6 `strip`

`strip` 方法返回去除两侧(不包括内部)空格的字符串：

```
>>> " internal whitespace is kept ".strip()
'internal whitespace is kept'
```

它和 `lower` 方法一起使用的话就可以很方便的对比输入的和存储的值。让我们回到 `lower` 部分中的用户名的例子，假设用户在输入名字时无意中在名字后面加上了空格：

```
>>> names = ["gumby", "smith", "jones"]
>>> name = "gumby "
>>> if name in names:
        print "Found it!"
...
>>> if name.strip() in names:
        print "Found it!"
...
Found it!
```

也可以指定需要去除的字符，将它们列为参数即可。

```
>>> "*** SPAM * for * everyone!!! ***".strip(" *!")
'SPAM * for * everyone'
```

这个方法只会去除两侧的字符，所以字符串中的星号没有被去掉。

附录B：`lstrip`、`rstrip`。

3.4.7 `translate`

`translate` 方法和 `replace` 方法一样，可以替换字符串中的某些部分，但是和前者不同的是，`translate` 方法只处理单个字符。它的优势在于可以同时进行多个替换，有些时候比 `replace` 效率高得多。

使用这个方法的方式有很多(比如替换换行符或者其他因平台而异的特殊字符)。但是让我们考虑一个简单的例子(很简单的例子)：假设需要将纯正的英文文本转换为带有德国口音的版本。为此，需要把字符 `c` 替换为 `k` 把 `s` 替换为 `z`。

在使用 `translate` 转换之前，需要先完成一张转换表。转换表中是以某字符替换某字符的对应关系。因为这个表(事实上是字符串)有多达256个项目，我们还是不要自己写了，使用 `string` 模块里面的 `maketrans` 函数就行。

`maketrans` 函数接受两个参数：两个等长的字符串，表示第一个字符串中的每个字符都用第二个字符串中相同位置的字符替换。明白了吗？来看一个简单的例子，代码如下：

```
>>> from string import maketrans
>>> table = maketrans("cs", "kz")
```

转换表中都有什么

转换表是包含替换ASCII字符集中256个字符的替换字母的字符串。

```
>>> table = maketrans("cs", "kz")
>>> len(table) 256
>>> table[97:123]
'abkdefghijklmnopqrztuvwxyz'
>>> maketrans("", "")[97:123]
'abcdefghijklmnopqrstuvwxyz'
```

正如你看到的，我已经把小写字母部分的表提取出来了。看一下这个表和空转换(没有改变任何东西)中的字母表。空转换包含一个普通的字母表，而在它前面的代码中，字母 `c` 和 `s` 分别被替换为 `k` 和 `z` 。

创建这个表以后，可以将它用作 `translate` 方法的参数，进行字符串的转换如下：

```
>>> "this is an incredible test".translate(table)
'thiz iz an inkredible tezt'
```

`translate` 的第二个参数是可选的，这个参数是用来指定需要删除的字符。例如，如果想要模拟一句语速超快的德国语，可以删除所有空格：

```
>>> "this is an incredible test".translate(table, " ") 'thizizaninkredibletezt'
```

请参见： `replace` 、 `lower` 。

3.5 小结

本章介绍了字符串的两种非常重要的使用方式。

字符串格式化：求模操作符(`%`)可以用来将其他值转换为包含转换标志的字符串，例如 `%s` 。它还能用来对值进行不同方式的格式化，包括左右对齐、设定字段宽度以及精度值，增加符号(正负号)或者左填充数字 `0` 等。

字符串方法：字符串有很多方法。有些非常有用(比如 `split` 和 `join`)，有些则用的很少(比如 `istitle` 或者 `capitalize`)。

3.5.1 本章的新函数

本章新涉及的函数如表3-2所示。

表3-2 本章的新函数

```
string.capwords(s[, sep])    使用split函数分隔字符串s(以sep为分隔符)，使用capitalize函数将分割得到的各单词首字母大写，并且使用join函数以sep为分隔符将各单词连接起来。
string.maketrans(from, to)   创建用于转换的转换表。
```

3.5.2 接下来学什么

列表、字符串和字典是Python中最重要的3种数据类型。列表和字符串已经学习过了，那么下面是什么呢？下一章中的主要内容是字典，以及字典如何支持索引以及其他方式的键(比如字符串和元组)。字典也提供了一些方法，但是数量没有字符串多。

第四章 字典：当索引不好用时

> 来源：http://www.cnblogs.com/Marlowes/p/5320049.html
>
> 作者：Marlowes

我们已经了解到，列表这种数据结构适合于将值组织到一个结构中，并且通过编号对其进行引用。在本章中，你将学到一种通过名字来引用值的数据结构。这种类型的数结构成为映射(mapping)。字典是Python中唯一内建的映射类型。字典中的值并没有特殊的顺序，但是都存储在一个特定的键(Key)下。键可以是数字、字符串甚至是元组。

4.1 字典的使用

字典这个名称已经给出了有关这个数据结构功能的一些提示：一方面，对于普通的书来说，都是按照从头到尾的顺序进行阅读。如果愿意，也可以快速翻到某一页，这有点像Python的列表。另一方面，构造字典的目的，不管是现实中的字典还是在Python中的字典，都是为了可以通过轻松查找某个特定的词语(键)，从而找到它的定义(值)。

某些情况下，字典比列表更加适用，比如：

√ 表示一个游戏棋盘的状态，每个键都是由坐标值组成的元组；

√ 存储文件修改时间，用文件名作为键；

√ 数字电话/地址簿。

假如有一个人名列表如下：

```
>>> names = ["Alice", "Beth", "Cecil", "Dee-Dee", "Earl"]
```

如果要创建一个可以存储这些人的电话号码的小型数据库，应该怎么做呢？一种方法是建立一个新的列表。假设只存储四位的分机电话号码，那么可以得到与下面相似的列表：

```
>>> numbers = ["2341", "9102", "3158", "0142", "5551"]
```

建立了这些列表后，可以通过如下方式查找Cecil的电话号码：

```
>>> numbers[names.index("Cecil")]
'3158'
```

这样做虽然可行，但是并不实用。真正需要的效果应该类似以下面这样：

```
>>> phonebook["Cecil"]
'3158'
```

你猜怎么着？如果`phonebook`是字典，就能像上面那样操作了。

整数还是数字字符串

看到这里，读者可能会有疑问：为什么用字符串而不用整数表示电话号码呢？考虑一下Dee-Dee的电话号码会怎么样：

```
>>> 0142
98
```

这并不是我们想要的结果，是吗？就像第一章曾经简略地提到的那样，八进制数字均以`0`开头。不能像那样表示十进制数字。

```
>>> 0912 File "<stdin>", line 1 0912
       ^ SyntaxError: invalid token
```

教训就是：电话号码(以及其他可能以`0`开头的数字)应该表示为数字字符串，而不是整数。

4.2 创建和使用字典

字典可以通过下面的方式创建：

```
>>> phonebook = {"Alice": "2341", "Beth": "9102", "Cecil": "3258"}
```

字典由多个键及与其对应的值构成的键-值对组成(我们也把键-值对称为项)。在上例中，名字是键，电话号码是值。每个键和它的值之间用冒号(`:`)隔开，项之间用逗号(`,`)隔开，而整个字典是由一对大括号括起来。空字典(不包括任何项)由两个大括号组成，像这样：`{}`。

注：字典中的键是唯一的*(*其他类型的映射也是如此*)*，而值并不唯一。

4.2.1 `dict` 函数

可以用`dict`函数(`dict`函数根本不是真正的函数，它是个类型，就像`list`、`tuple`和`str`一样)，通过其他映射(比如其他字典)或者(键，值)对的序列建立字典。

```
>>> items = [("name", "Gumby"), ("age", 42)]
>>> d = dict(items) >>> d
{'age': 42, 'name': 'Gumby'}
>>> d["name"] 'Gumby'
```

`dict` 函数也可以通过关键字参数来创建字典，如下例所示：

```
>>> d = dict(name="Gumby", age=42)
>>> d
{'age': 42, 'name': 'Gumby'}
```

尽管这可能是 `dict` 函数最有用的功能，但是还能以映射作为 `dict` 函数的参数，以建立其项与映射相同的字典(如果不带任何参数，则 `dict` 函数返回一个新的空字典，就像 `list` 、 `tuple` 以及 `str` 等函数一样)。如果另一个映射也是字典(毕竟这是唯一内建的映射类型)，也可以使用本章稍后讲到的字典方法 `copy` 。

4.2.2 基本字典操作

字典的基本行为在很多方面与序列(sequence)类似：

√ `len(d)` 返回 `d` 中项(键-值对)的数量；

√ `d[k]` 返回关联到键 `k` 上的值；

√ `d[k]=v` 将值 `v` 关联到键k上；

√ `del d[k]` 删除键为 `k` 的项；

√ `k in d` 检查 `d` 中是否有含有键为k的项。

尽管字典和列表有很多特性相同，但也有下面一些重要的区别。

√键类型：字典的键不一定为整型数据(但也可以是)，键可以是任意的不可变类型，比如浮点型(实型)、字符串或者元组。

√自动添加：即使键起初在字典中并不存在，也可以为它赋值，这样字典就会建立新的项。而(在不使用 `append` 方法或者其他类似操作的情况下)不能将值关联到列表范围之外的索引上。

√成员资格：表达式 `k in d` (`d` 为字典)查找的是键，而不是值。表达式 `v in l` (`l` 为列表)则用来查找值，而不是索引。这样看起来好像有些不太一致，但是当习惯以后就会感觉非常自然了。毕竟，如果字典含有指定的键，查找相应的值也就很容易了。

注：在字典中检查键的成员资格比在列表中检查值的成员资格更高效，数据结构的规模越大，两者的效率差距越明显。

第一点——键可以是任意不可变类型——是字典最强大的地方。第二点也很重要。看看下面的区别：

```
>>> x = []  # 列表
>>> x[42] = "Foobar" Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  IndexError: list assignment index out of range
>>> x = {}  # 字典
>>> x[42] = "Foobar"
>>> x
{42: 'Foobar'}
```

首先，程序试图将字符串`"Foobar"`关联到一个空列表的42号位置上——这显然是不可能的，因为这个位置根本不存在。为了将其变为可能，我必须用`[None]*43`或者其他方式初始化`x`，而不能仅使用`[]`。但是下一个例子工作得很好。我将`"Foobar"`关联到空字典的键`42`上，没问题！新的项已经添加到字典中，我达到目的了。

代码清单4-1所示是电话本例子的代码。

```
 1 #!/usr/bin/env python
 2 # coding=utf-8
 3
 4 # 一个简单的数据库
 5 # 字典使用人名作为键。每个人用另一个字典来表示，其键"phone"和"addr"分别表示他们的电话号码和地址。
 6
 7 people = {
 8
 9     "Alice": {
10         "phone": "2341",
11         "addr": "Foo drive 23"
12 },
13
14     "Beth": {
15         "phone": "9102",
16         "addr": "Bar street 42"
17 },
18
19     "Cecil": {
20         "phone": "3158",
21         "addr": "Baz avenue 90"
22 }
23 }
24
25 # 针对电话号码和地址使用的描述性标签，会在打印输出的时候用到
26 labels = {
27     "phone": "phone number",
28     "addr": "address"
29 }
30
31 name = raw_input("Name: ")
32
33 # 查找电话号码还是地址
34 request = raw_input("Phone number (p) or address (a)? ")
35
36 # 使用正确的键
37 if request == "p":
38     key = "phone"
39 if request == "a":
40     key = "addr"
41
42 # 如果名字是字典中的有效键才打印信息
43 if name in people:
44     print "%s's %s is %s." % (name, labels[key], people[name][key])
```

Code_Listing 4-1

下面是程序的运行示例：

```
Name: Beth
Phone number (p) or address (a)? a
Beth's address is Bar street 42.
```

4.2.3 字典的格式化字符串

在第三章，已经见过如何使用字符串格式化功能来格式化元组中所有的值。如果使用的是字典(只以字符串作为键的)而不是元组，会使字符串格式化更酷一些。在每个转换说明符(conversion specifier)中的%字符后面，可以加上键(用圆括号括起来)，后面再跟上其他说明元素。

```
>>> phonebook
{'Beth': '9102', 'Alice': '2341', 'Cecil': '3258'}
>>> "Cecil's phone number is %(Cecil)s." % phonebook
"Cecil's phone number is 3258."
```

除了增加的字符串键之外，转换说明符还是像以前一样工作。当以这种方式使用字典的时候，只要所有给出的键都能在字典中找到，就可以使用任意数量的转换说明符。这类字符串格式化在模板系统中非常有用(本例中使用HTML)。

```
>>> template = """<html>
... <head><title>%(title)s</title></head>
... <body>
... <h1>%(title)s</h1>
... <p>%(text)s</p>
... </body>"""
>>> data = {"title": "My Home Page", "text": "Welcome to my home page!"}
>>> print template % data
<html>
<head><title>My Home Page</title></head>
<body>
<h1>My Home Page</h1>
<p>Welcome to my home page!</p>
</body>
```

注：`string.Template`类(第三章提到过)对于这类应用也是非常有用的。

4.2.4 字典方法

就像其他内建类型一样，字典也有方法。这些方法非常有用，但是可能不会像列表或者字符串方法那样被频繁地使用。读者最好先简单浏览一下本节，了解有哪些方法可用，然后在需要的时候再回过头来查看特定方法的具体用法。

1. `clear`

`clear`方法清除字典中所有的项。这是个原地操作(类似于`list.sort`)，所以无返回值(或者说返回`None`)。

```
>>> d = {} >>> d["name"] = "Gumby"
>>> d["age"] = 42
>>> d
{'age': 42, 'name': 'Gumby'}
>>> returned_value = d.clear()
>>> d
{}
>>> print returned_value
None
```

为什么这个方法有用呢？考虑以下两种情况。

```
>>> x = {}  # 第一种情况
>>> y = x
>>> x["key"] = "value"
>>> y
{'key': 'value'}
>>> x = {}
>>> y
{'key': 'value'}
>>> x = {}  # 第二种情况
>>> y = x
>>> x["key"] = "value"
>>> y
{'key': 'value'}
>>> x.clear() >>> y
{}
```

两种情况中， `x` 和 `y` 最初对应同一个字典。情况1中，我通过将 `x` 关联到一个新的空字典来“清空”它，这对 `y` 一点影响也没有，它还关联到原先的字典。这可能是所需要的行为，但是如果真的想清空原始字典中的所有的元素，必须使用 `clear` 方法。正如在情况2中所看到的， `y` 随后也被清空了。

2. copy

`copy` 方法返回一个具有相同键-值对的新字典(这个方法实现的是浅复制(shallow copy)，因为值本身就是相同的，而不是副本)。

```
>>> x = {"username": "admin", "machines": ["foo", "bar", "baz"]}
>>> y = x.copy() >>> y["username"] = "mlh"
>>> y["machines"].remove("bar")
>>> y
{'username': 'mlh', 'machines': ['foo', 'baz']}
>>> x
{'username': 'admin', 'machines': ['foo', 'baz']}
```

可以看到，当在副本中替换值的时候，原始字典不受影响，但是，如果修改了某个值(原地修改，而不是替换)，原始的字典也会改变，因为同样的值也存储在原字典中(就像上面例子中的 `machines` 列表一样)。

避免这种问题的一种方法就是使用深复制(deep copy)，复制其包含的所有值。可以使用 `copy` 模块的 `deepcopy` 函数来完成操作：

```
>>> from copy import deepcopy
>>> d = {}
>>> d["names"] = ["Alfred", "Bertrand"]
>>> c = d.copy()
>>> dc = deepcopy(d)
>>> d["names"].append("Clive")
>>> c
{'names': ['Alfred', 'Bertrand', 'Clive']}
>>> dc
{'names': ['Alfred', 'Bertrand']}
```

3. `fromkeys`

`fromkeys` 方法使用给定的键建立新的字典，每个键都对应一个默认的值 `None` 。

```
>>> {}.fromkeys(["name", "age"])
{'age': None, 'name': None}
```

刚才的例子中首先构造了一个空字典，然后调用它的 `fromkeys` 方法，建立另外一个字典——有些多余。此外，你还可以直接在 `dict` 上面调用该方法，前面讲过， `dict` 是所有字典的类型(关于类型和类的概念在第七章中会深入讨论)。

```
>>> dict.fromkeys(["name", "age"])
{'age': None, 'name': None}
```

如果不想使用 `None` 作为默认值，也可以自己提供默认值。

```
>>> dict.fromkeys(["name", "age"], "(unknown)")
{'age': '(unknown)', 'name': '(unknown)'}
```

4. `get`

`get` 方法是个更宽松的访问字典项的方法。一般来说，如果试图访问字典中不存在的项时会出错：

```
>>> d = {} >>> print d["name"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  KeyError: 'name'
```

而用 `get` 就不会：

```
>>> print d.get("name")
None
```

可以看到，当使用 `get` 访问一个不存在的键时，没有任何异常，而得到了 `None` 值。还可以自定义“默认”值，替换 `None` ：

```
>>> d.get("name", "N/A") 'N/A'
```

如果键存在，`get` 使用起来就像普通的字典查询一样：

```
>>> d["name"] = "Eric"
>>> d.get("name") 'Eric'
```

代码清单4-2演示了一个代码清单4-1程序的修改版本，它使用 `get` 方法访问"数据库"实体。

```
 1 #!/usr/bin/env python
 2 # coding=utf-8
 3
 4 # 一个简单的数据库
 5 # 字典使用人名作为键。每个人用另一个字典来表示，其键"phone"和"addr"分别表示他们的电话号码和地址。
 6
 7 people = {
 8
 9     "Alice": {
10         "phone": "2341",
11         "addr": "Foo drive 23"
12 },
13
14     "Beth": {
15         "phone": "9102",
16         "addr": "Bar street 42"
17 },
18
19     "Cecil": {
20         "phone": "3158",
21         "addr": "Baz avenue 90"
22 }
23 }
24
25 # 针对电话号码和地址使用的描述性标签，会在打印输出的时候用到
26 labels = {
27     "phone": "phone number",
28     "addr": "address"
29 }
30
31 name = raw_input("Name: ")
32
33 # 查找电话号码还是地址
34 request = raw_input("Phone number (p) or address (a)? ")
35
36 # 使用正确的键
37 key = request  # 如果请求既不是"p"也不是"a"
38
39 if request == "p": 40     key = "phone"
41 if request == "a": 42     key = "addr"
43
44 # 使用get()提供默认值
45 person = people.get(name, {})
46 label = labels.get(key, key)
47 result = person.get(key, "not available")
48
49 print "%s's %s is %s." % (name, label, result)
```

Code_Listing 4-2

以下是程序运行的输出。注意 `get` 方法带来的灵活性如何使得程序在用户输入我们并未准备的值时也能做出合理的反应。

```
Name: Gumby
Phone number (p) or address (a)? batting average
Gumby's batting average is not available.
```

5. `has_key`

`has_key` 方法可以检查字典中是否含有特定的键。表达式 `d.has_key(k)` 相当于表达式 `k in d` 。使用哪个方式很大程度上取决于个人的喜好。Python3.0中不包括这个函数。

下面是一个使用 `has_key` 方法的例子：

```
>>> d = {}
>>> d.has_key("name")
False
>>> d["name"] = "Eric"
>>> d.has_key("name")
True
```

6. `items` 和 `iteritems`

`items` 方法将字典所有的项以列表方式返回，列表中的每一项都表示为(键, 值)对的形式。但是项在返回时并没有遵循特定的次序。

```
>>> d = {"title": "Python Web Site", "url": "http://www.python.org", "spam": "0"}
>>> d.items()
[('url', 'http://www.python.org'), ('spam', '0'), ('title', 'Python Web Site')]
```

`iteritems` 方法的作用大致相同，但是会返回一个迭代器对象而不是列表：

```
>>> it = d.iteritems()
>>> it <dictionary-itemiterator object at 0x00000000029BAEF8>
>>> list(it)  # Convert the iterator to a list
[('url', 'http://www.python.org'), ('spam', '0'), ('title', 'Python Web Site')]
```

在很多情况下使用 `iteritems` 会更加高效(尤其是想要迭代结果的情况下)。关于迭代器的更多信息，请参见第九章。

7. `keys` 和 `iterkeys`

`keys` 方法将字典中的键以列表形式返回，而 `iterkeys` 则返回针对键的迭代器。

8. `pop`

`pop` 方法用来获得对应于给定键的值，然后将这个键-值对从字典中移除。

```
>>> d = {"x": 1, "y": 2}
>>> d.pop("x") 1
>>> d
{'y': 2}
```

9. `popitem`

`popitem` 方法类似于 `list.pop` ，后者会弹出列表的最后一个元素。但不同的是， `popitem` 弹出随机的项，因为字典并没有“最后的元素”或者其他有关顺序的概念。若想一个接一个地移除并处理项，这个方法就非常有效了(因为不用首先获取键的列表)。

```
>>> d
{'url': 'http://www.python.org', 'spam': '0', 'title': 'Python Web Site'}
>>> d.popitem()
('url', 'http://www.python.org')
>>> d
{'spam': '0', 'title': 'Python Web Site'}
```

尽管 `popitem` 和列表的 `pop` 方法很类似，但字典中没有与 `append` 等价的方法。因为字典是无序的，类似于 `append` 的方法是没有任何意义的。

10. `setdefault`

`setdefault` 方法在某种程度上类似于 `get` 方法，能够获得与给定键相关联的值，除此之外， `setdefault` 还能在字典中不含有给定键的情况下设定相应的键值。

```
>>> d = {}
>>> d.setdefault("name", "N/A") 'N/A'
>>> d
{'name': 'N/A'}
>>> d["name"] = "Gumby"
>>> d.setdefault("name", "N/A")
'Gumby'
>>> d
{'name': 'Gumby'}
```

可以看到，当键不存在的时候， `setdefault` 返回默认值并且相应地更新字典。如果键存在，那么就返回与其对应的值，但不改变字典。默认值是可选的，这点和 `get` 一样。如果不设定，会默认使用 `None` 。

```
>>> d = {}
>>> print d.setdefault("name")
None
>>> d
{'name': None}
```

11. `update`

`update` 方法可以利用一个字典项更新另外一个字典：

```
>>> d = {
... "title": "Python Web Site",
... "url": "http://www.python.org",
... "changed": "Mar 14 22:09:15 MET 2008" ...     }
>>> x = {"title": "Python Language Website"}
>>> d.update(x)
>>> d
{'url': 'http://www.python.org', 'changed': 'Mar 14 22:09:15 MET 2008', 'title': 'Pyth
on Language Website'}
```

提供的字典中的项会被添加到旧的字典中，若有相同的键则会进行覆盖。

`update` 方法可以使用与调用 `dict` 函数(或者类型构造函数)同样的方式进行调用，这点在本章前面已经讨论。这就意味着 `update` 可以和映射、拥有(键、值)对的队列(或者其他可迭代的对象)以及关键字参数一起调用。

12. `values` 和 `itervalues`

`values` 方法以列表的形式返回字典中的值(`itervalues` 返回值的迭代器)。与返回键的列表不同的是，返回值的列表中可以包含重复的元素：

```
>>> d = {}
>>> d[1] = 1
>>> d[2] = 2
>>> d[3] = 3
>>> d[4] = 1
>>> d.values()
[1, 2, 3, 1]
```

4.3 小结

本章介绍了如下内容。

映射：映射可以使用任意不可变对象标识元素。最常用的类型是字符串和元组。Python唯一的内建映射类型是字典。

利用字典格式化字符串：可以通过在格式化说明符中包括名称(键)来对字典应用字符串格式化操作。在当字符串格式化中使用元组时，还需要对元组中每一个元素都设定“格式化说明符”。在使用字典时，所用的说明符可以比在字典中用到的项少。

字典的方法：字典有很多方法，调用的方式和调用列表以及字符串方法的方式相同。

4.3.1 本章的新函数

本章涉及的新函数如表4-1所示。

表4-1 本章的新函数

dict(seq)	用(键、值)对(或者映射和关键字参数)建立字典。

4.3.2 接下来学什么

到现在为止，已经介绍了很多有关Python的基本数据类型的只是，并且讲解了如何使用它们来建立表达式。那么请回想一下第一章的内容，计算机程序还有另外一个重要的组成因素——语句。下一章我们会对语句进行详细的讨论。

第五章 条件、循环和其他语句

> 来源：http://www.cnblogs.com/Marlowes/p/5329066.html
>
> 作者：Marlowes

读者学到这里估计都有点不耐烦了。好吧，这些数据结构什么的看起来都挺好，但还是没法用它们做什么事，对吧？

下面开始，进度会慢慢加快。前面已经介绍过了几种基本语句(`print`语句、`import`语句、赋值语句)。在深入介绍条件语句和循环语句之前，我们先来看看这几种基本语句更多的使用方法。随后你会看到列表推倒式(list comprehension)如何扮演循环和条件语句的角色——尽管它本身是表达式。最后介绍`pass`、`del`和`exec`语句的用法。

5.1 `print` 和 `import` 的更多信息

随着更加深入第学习Python，可能会出现这种感觉：有些自己以为已经掌握的知识点，还隐藏着一些让人惊讶的特性。首先来看看`print`(在Python3.0中，`print`不再是语句——而是函数(功能基本不变))和`import`的几个比较好的特性。

注：对于很多应用程序来说，使用*`logging`*模块记日志比*`print`*语句更合适。更多细节请参见第十九章。

5.1.1 使用逗号输出

前面的章节中讲解过如何使用`print`来打印表达式——不管是字符串还是其他类型进行自动转换后的字符串。但是事实上打印多个表达式也是可行的，只要将它们用逗号隔开就好：

```
>>> print "Age", 19
Age 19
```

可以看到，每个参数之间都插入了一个空格符。

注：*print*的参数并不能像我们预期那样构成一个元组：

```
>>> 1, 2, 3
(1, 2, 3)
>>> print 1, 2, 3
1 2 3
>>> print (1, 2, 3)
(1, 2, 3)
```

如果想要同时输出文本和变量值，却又不希望使用字符串格式化的话，那这个特性就非常有用了：

```
>>> name = "XuHoo"
>>> salutation = "Mr."
>>> greeting = "Hello,"
>>> print greeting, salutation, name
Hello, Mr. XuHoo
# 注意，如果greeting字符串不带逗号，那么结果中怎么能得到逗号呢？像下面这样做是不行的：
>>> print greeting, ",", salutation, name
Hello , Mr. XuHoo
# 因为上面的语句会在逗号前加入空格。下面是一种解决方案：
>>> print greeting + ",", salutation, name
Hello, Mr. XuHoo
# 这样一来，问候语后面就只会增加一个逗号。
```

如果在结尾处加上逗号，那么接下来的语句会与前一条语句在同一行打印，例如：

```
print "Hello", print "world!"

# 输出 Hello, world!(这只在脚本中起作用，而在交互式Python会话中则没有效果。在交互式会话中，所有的语句都会被单独执行(并且打印出内容))
```

5.1.2 把某件事作为另一件事导入

从模块导入函数的时候，通常可以使用以下几种方式：

```
import somemodule # or
from somemodule import somefunction
# or
from somemodule import somefunction, anotherfunction, yetanotherfunction
# or
from somemodule import *
```

只有确定自己想要从给定的模块导入所有功能时，才应该使用最后一个版本。但是如果两个模块都有 `open` 函数，那又该怎么办？只需要使用第一种方式导入，然后像下面这样使用函数：

```
import module1 import module2

module1.open(...)
module2.open(...)
```

但还有另外的选择：可以在语句末尾增加一个 `as` 子句，在该子句后给出想要使用的别名。例如为整个模块提供别名：

```
>>> import math as foobar
>>> foobar.sqrt(4)
2.0

# 或者为函数提供别名
>>> from math import sqrt as foobar
>>> foobar(4)
2.0

# 对于open函数，可以像下面这样使用：
from module1 import open as open1
from module2 import open as open2
```

注：有些模块，例如 `os.path` 是分层次安排的(一个模块在另一个模块的内部)。有关模块结构的更多信息，请参见第十章关于包的部分。

5.2 赋值魔法

就算是不起眼的赋值语句也有一些特殊的技巧。

5.2.1 序列解包

赋值语句的例子已经给过不少，其中包括对变量和数据结构成员的(比如列表中的位置和分片以及字典中的槽)赋值。但赋值的方法还不止这些。比如，多个赋值操作可以同时进行：

```
>>> x, y, z = 1, 2, 3
>>> print x, y, z 1 2 3

# 很有用吧？用它交换两个(或更多个)变量也是没问题的：
>>> x, y = y, x >>> print x, y, z 2 1 3
```

事实上，这里所做的事情叫做序列解包(sequence unpacking)或递归解包——将多个值的序列解开，然后放到变量的序列中。更形象一点的表示就是：

```
>>> values = 1, 2, 3
>>> values
(1, 2, 3) >>> x, y, z  = values >>> x 1
>>> y 2
>>> z 3
```

当函数或者方法返回元组(或者其他序列或可迭代对象)时，这个特性尤其有用。假设需要获取(和删除)字典中任意的键-值对，可以使用 `popitem` 方法，这个方法将键-值作为元组返回。那么这个元组就可以直接赋值到两个变量中：

```
>>> scoundrel = {"name": "XuHoo", "girlfriend": "None"}
# =_=
>>> key, value = scoundrel.popitem()
>>> key 'girlfriend'
>>> value 'None'
```

它允许函数返回一个以上的值并且打包成元组，然后通过一个赋值语句很容易进行访问。所解包的序列中的元素数量必须和放置在赋值符号=左边的变量数量完全一致，否则Python会在赋值时引发异常：

```
>>> x, y, z = 1, 2 Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  ValueError: need more than 2 values to unpack
>>> x, y, z = 1, 2, 3, 4
  Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  ValueError: too many values to unpack
```

注：*Python3.0*中有另外一个解包的特性：可以像在函数的参数列表中一样使用星号运算符*(*参见第六章*)*。例如，`a, b, *rest = [1, 2, 3, 4]`最终会在`a`和`b`都被赋值之后将所有的其他的参数都收集到`rest`中。本例中，`rest`的结果将会是`[3, 4]`。使用星号的变量也可以放在第一个位置，这样它就总会包含一个列表。右侧的赋值语句可以是可迭代对象。

5.2.2 链式赋值

链式赋值(charned assignment)是将同一个值赋给多个变量的捷径。它看起来有些像上节中并行赋值，不过这里只处理一个值：

```
x = y = somefunction() # 和下面语句的效果是一样的：
y = somefunction()
x = y
# 注意上面的语句和下面的语句不一定等价：
x = somefunction()
y = somefunction()
```

有关链式赋值更多的信息，请参见本章中的“同一性运算符”一节。

5.2.3 增量赋值

这里没有将赋值表达式写为`x=x+1`，而是将表达式运算符(本例中是`±`)放置在赋值运算符`=`的左边，写成`x+=1`,。这种写法叫做增量赋值(augmented assignmnet)，对于`*`、`/`、`%`等标准运算符都适用：

```
>>> x = 2
>>> x += 1
>>> x *= 2
>>> x 6

# 对于其他数据类型也适用(只要二元运算符本身适用于这些数据类型即可)：
>>> fnord = "foo"
>>> fnord += "bar"
>>> fnord *= 2
>>> fnord 'foobarfoobar'
```

增量赋值可以让代码更加紧凑和简练，很多情况下会更易读。

5.3 语句块：缩排的乐趣

语句块并非一种语句，而是在掌握后面两节的内容之前应该了解的知识。

语句块是在条件为真(条件语句)时执行或者执行多次(循环语句)的一组语句。在代码前放置空格来缩进语句即可创建语句块。

注：使用`tab`字符也可以缩进语句块。*Python*将一个`tab`字符解释为到下一个`tab`字符位置的移动，而一个`tab`字符位置为*8*个空格，但是标准且推荐的方式是只用空格，尤其是在每个缩进需要*4*个空格的时候。

块中的每行都应该缩进同样的量。下面的伪代码(并非真正Python代码)展示了缩进的工作方法：

```
this is a line
this is another line:
    this is another block
    continuing the same block
    the last line of this block
phew, there we escaped the inner block
```

很多语言使用特殊单词或者字符(比如`begin`或`{`)来表示一个语句块的开始，使用另外的单词或者字符(比如`end`或者`}`)表示语句块的结束。在Python中，冒号(`:`)用来标识语句块的开始，块中的每一个语句都是缩进(缩进量相同)。当回退到和已经闭合的块一样的缩进量时，就表示当前块已经结束了(很多程序编辑器和集成开发环境都知道如何缩进语句块，可以帮助用户轻松把握缩进)。

现在我确信你已经等不及想知道语句块怎么使用了。废话不多说，我们来看一下。

5.4 条件和条件语句

到目前为止的程序都是一条一条语句顺序执行的。在这部分中会介绍让程序选择是否执行语句块的方法。

5.4.1 这就是布尔变量的作用

真值(也叫作布尔值，这个名字根据在真值上做过大量研究的George Boole命名的)是接下来内容的主角。

注：如果注意力够集中，你就会发现在第一章的“管窥：`if`语句”中就已经描述过`if`语句。到目前为止这个语句还没有被正式介绍。实际上，还有很多`if`语句的内容没有介绍。

下面的值在作为布尔表达式的时候，会被解释器看做假(`False`)：

```
False    None    0    ""    ()    []    {}
```

换句话说，也就是标准值 `False` 和 `None` 、所有类型的数字 `0` (包括浮点型、长整型和其他类型)、空序列(比如空字符串、元组和列表)以及空的字典都为假。其他的一切(至少当我们讨论内建类型是是这样——第九章内会讨论构建自己的可以被解释为真或假的对象)都被解释为真，包括特殊值 `True` (Python经验丰富的Laura Creighton解释说这个区别类似于“有些东西”和“没有东西”的区别，而不是真和假的区别)。

明白了吗？也就是说Python中的所有值都能被解释为真值，初次接触的时候可能会有些搞不明白，但是这点的确非常有用。“标准的”布尔值为 `True` 和 `False` 。在一些语言中(例如C和Python2.3以前的版本)，标准的布尔值为 `0` (表示假)和 `1` (表示真)。事实上， `True` 和 `False` 只不过是 `1` 和 `0` 的一种“华丽”的说法而已——看起来不同，但作用相同。

```
>>> True
True
>>> False
False
>>> True == 1
True
>>> False == 0
True
>>> True + False
1
>>> True + False + 19
20
```

那么，如果某个逻辑表达式返回 `1` 或 `0` (在老版本Python中)，那么它实际的意思是返回 `True` 或 `False` 。

布尔值 `True` 和 `False` 属于布尔类型， `bool` 函数可以用来(和 `list` 、 `str` 以及 `tuple` 一样)转换其他值。

```
>>> bool("I think, therefore I am")
True
>>> bool(19)
True
>>> bool("")
False
>>> bool(0)
False
```

因为所有值都可以用作布尔值，所以几乎不需要对它们进行显示转换(可以说Python会自动转换这些值)。

注：尽管 `[]` 和 `""` 都是假肢(也就是说 `bool([])==bool("")==False`)，它们本身却并不相等(也就是说 `[]!=""`)。对于其他不同类型的假值对象也是如此(例如 `()!=False`)。

5.4.2 条件执行和 `if` 语句

真值可以联合使用(马上就要介绍)，但还是让我们先看看它们的作用。试着运行下面的脚本：

```
name = raw_input("What is your name? ")
if name.endswith("XuHoo"):
    print "Hello, Mr.XuHoo"
```

这就是 `if` 语句，它可以实现条件执行，即如果条件(在 `if` 和冒号之间的表达式)判定为眞，那么后面的语句块(本例中是单个 `print` 语句)就会被执行。如果条件为假，语句块就不会被执行(你猜到了，不是吗)。

注：在第一章的“管窥： `if` 语句”中，所有语句都写在一行中。这种书写方式和上例中的使用单行语句块的方式是等价的。

5.4.3 else子句

前一节的例子中，如果用户输入了以 `XuHoo` 作为结尾的名字，那么 `name.endswit` 方法就会返回眞，使得 `if` 进入语句块，打印出问候语。也可以使用 `else` 子句增加一种选择(之所以叫做子句是因为它不是独立的语句，而只能作为 `if` 语句的一部分)。

```
name = raw_input("What is your name? ")
if name.endswith("XuHoo"):
    print "Hello, Mr.XuHoo"
else:
    print "Hello. stranger"
```

如果第一个语句块没有被执行(因为条件被判定为假)，那么就会站转入第二个语句块，可以看到，阅读Python代码很容易，不是吗？大声把代码读出来(从 `if` 开始)，听起来就像正常(也可能不是很正常)句子一样。

5.4.4 elif子句

如果需要检查多个条件，就可以使用 `elif` ，它是 `else if` 的简写，也是 `if` 和 `else` 子句的联合使用，也就是具有条件的 `else` 子句。

```
name = input("Enter a number: ")
if num > 0:
    print "The number is positive"
elif num < 0:
    print "The number is negative"
else:
    print "The number is zero"
```

注：可以使用 *`int(raw_input(...))`* 函数来代替 *`input(...)`* 。关于两者的区别，请参见第一章。

5.4.5 嵌套代码块

下面的语句中加入了一些不必要的内容。if语句里面可以嵌套使用 `if` 语句，就像下面这样：

```
name = raw_input("What is your name? ")
if name.endswith("XuHoo"):
    if name.startswith("Mr."):
        print "Hello, Mr. XuHoo"
    elif name.startswith("Mrs."):
        print "Hello, Mrs. XuHoo"
    else:
        print "Hello, XuHoo"
else:
    print "Hello, stranger"
```

如果名字是以 `XuHoo` 结尾的话，还要检查名字的开头——在第一个语句块中的单独的 `if` 语句中。注意这里 `elif` 的使用。最后一个选项中(`else` 子句)没有条件——如果其他的条件都不满足就使用最后一个。可以把任何一个 `else` 子句放在语句块外面。如果把里面的 `else` 子句放在外面的话，那么不以 `Mr.` 或 `Mrs.` 开头(假设这个名字是 `XuHoo`)的名字都被忽略掉了。如果不写最后一个 `else` 子句，那么陌生人就被忽略掉。

5.4.6 更复杂的条件

以上就是有关if语句的所有知识。下面让我们回到条件本身，因为它们才是条件执行时真正有趣的部分。

1. 比较运算符

用在条件中的最基本的运算符就是比较运算符了，它们用来比较其他对象。比较运算符已经总结在表5-1中。

表5-1 Python中的比较运算符

```
x = y                     x 等于 y
x < y                      x 小于 y
x > y                      x 大于 y
x >= y                      x 大于等于 y
x <= y                     x 小于等于 y
x != y                    x 不等于 y
x is y                    x 和 y 是同一个对象
x is not y                 x 和 y 是不同的对象
x in y                    x 是 y 容器(例如，序列)的成员
x not in y                 x 不是 y 容器(例如，序列)的成员
```

比较不兼容类型

理论上，对于相对大小的任意两个对象 `x` 和 `y` 都是可以使用比较运算符(例如， `<` 和 `<=`)比较的，并且都会得到一个布尔值结果。但是只有在 `x` 和 `y` 是相同或者近似类型的对象时，比较才有意义(例如，两个整型数或者一个整型数和一个浮点型数进行比较)。

正如将一个整型数添加到一个字符串中是没有意义的，检查一个整型是否比一个字符串小，看起来也是毫无意义的。但奇怪的是，在Python3.0之前的版本中这却是可以的。对于此类比较行为，读者应该敬而远之，因为结果完全不可靠，在每次程序执行的时候得到的结果都可能不同。在Python3.0中，比较不兼容类型的对象已经不再可行。

注：如果你偶然遇见 `x <> y` 这样的表达式，它的意思其实就是 `x != y` 。不建议使用 `<>` 运算符，应该尽量避免使用它。

在Python中比较运算符和赋值运算符一样是可以连接的——几个运算符可以连在一起使用，比如： `0<age<100` 。

注：比较对象的时候可以使用第二章中介绍的内建的 `cmp` 函数。

有些运算符值得特别关注，下面的章节中会对此进行介绍。

2. 相等运算符

如果想要知道两个东西是否相等，应该使用相等运算符，即两个等号"=="：

```
>>> "foo" == "foo" True
>>> "foo" == "bar" False
# 相等运算符需要使用两个等号，如果使用一个等号会出现下面的情况
>>> "foo" = "foo"
    File "<stdin>", line 1
    SyntaxError: can't assign to literal
```

单个相等运算符是赋值运算符，是用来改变值的，而不能用来比较。

3. `is` ：同一性运算符

这个运算符比较有趣。它看起来和 `==` 一样，事实上却不同：

```
>>> x = y = [1, 2, 3]
>>> z = [1, 2, 3]
>>> x == y
True >>> x == z
True >>> x is y
True >>> x is z
False
```

到最后一个例子之前，一切看起来都很好，但是最后一个结果很奇怪， `x` 和 `z` 相等却不等同，为什么呢？因为 `is` 运算符是判定同一性而不是相等性的。变量 `x` 和 `y` 都被绑定到同一列表上，而变量 `z` 被绑定在另外一个具有相同数值和顺序的列表上。它们的值可能相等，但是却不是同一个对象。

这看起来有些不可理喻吧？看看这个例子：

```
>>> x = [1, 2, 3]
>>> y = [2, 4]
>>> x is not y
True
>>> del x[2]
>>> y[1] = 1
>>> y.reverse()
>>> y
[1, 2]
>>> x
[1, 2] # 本例中，首先包括两个不同的列表x和y。可以看到 x is not y 与(x is y 相反)，这个已经知道了
。之后我改动了一下列表，尽管它们的值相等了，但是还是两个不同的列表。
>>> x == y
True
>>> x is y
False # 显然，两个列表值等但是不等同。
```

总结一下：使用 `==` 运算符来判定两个对象是否相等。使用 `is` 判定两者是否等同(同一个对象)。

注：避免将 *`is`* 运算符用于比较类似数值和字符串这类不可变值。由于*Python*内部操作这些对象的方式的原因，使用 *`is`* 运算符的结果是不可预测的。

4. `in` ：成员资格运算符

`in` 运算符已经介绍过了(在2.2.5节)。它可以像其他比较运算符一样在条件语句中使用。

```
name = raw_input("What is your name? ")
if "s" in name:
    print "Your name contains the letter 's'."
else:
    print "Your name does not contains the letter 's'."
```

5. 字符串和序列比较

字符串可以按照字母顺序排列进行比较。

```
>>> "alpha" < "beta" True
```

注：实际的顺序可能会因为使用不同的本地化设置*(`locale`)*而和上边的例子有所不同*(*请参见标准库文档中 *`locale`* 模块一节*)*。

如果字符串内包括大写字母，那么结果就会有点乱(实际上，字符是按照本身的顺序值排列的。一个字母的顺序值可以用 `ord` 函数查到， `ord` 函数与 `chr` 函数功能相反)。如果要忽略大小写字母的区别，可以使用字符串方法 `upper` 和 `lower` (请参见第三章)。

```
>>> "FnOrD".lower() == "Fnord".lower()
True # 其他的序列也可以用同样的方式进行比较，不过比较的不是字符而是其他类型的元素。
>>> [1, 2] < [2 ,1]
True # 如果一个序列中包括其他序列元素，比较规则也同样适用于序列元素。
>>> [2, [1, 4]] < [2, [1, 5]]
True
```

6. 布尔运算符

返回布尔值的对象已经介绍过许多(事实上，所有值都可以解释为布尔值，所有的表达式也都返回布尔值)。但有时想要检查一个以上的条件。例如，如果需要编写读取数字并且判断该数字是否位于1~10之间(也包括10)的程序，可以像下面这样做：

```
number = input("Enter a number between 1 and 10: ")
if number <= 10:
    if number >= 1:
        print "Great!"
    else:
        print "Wrong!"
else:
    print "Wrong!"

# 这样做没问题，但是方法太笨了。笨在需要写两次print "Wrong!"。在复制上浪费精力可不是好事。那么怎么办？很简单：
number = input("Enter a number between 1 and 10: ")
if number <= 10 and number >= 1:
    print "Great!"
else:
    print "Wrong!"
```

注：本例中，还有*(或者说应该使用)*更简单的方法，即使用连接比较：`1<=number<=10`。

`and` 运算符就是所谓的布尔运算符。它连接两个布尔值，并且在两者都为真时返回真，否则返回假。与它同类的还有两个运算符，`or` 和 `not`。使用这三个运算符就可以随意结合真值。

```
if ((cash > price) or customer_has_good_credit) and not out_of_stock:
    give_goods()
```

短路逻辑和条件表达式

布尔运算符有个有趣的特性：只有在需要求值时才进行求值。举例来说，表达式 `x and y` 需要两个变量都为真时才为真，所以如果x为假，表达式就会立刻返回 `False`，而不管 `y` 的值。实际上，如果 `x` 为假，表达式会返回 `x` 的值——否则它就返回 `y` 的值。(能明白它是怎么达到预期效果的吗？)这种行为被称为短路逻辑(short-circuit logic)或惰性求值(lazy evaluation)：布尔运算符通常被称为逻辑运算符，就像你看到的那样第二个值有时“被短路了”。这种行为对于 `or` 来说也同样适用。在 `x or y` 中，`x` 为真时，它直接返回 `x` 值，否则返回 `y` 值。(应该明白什么意思吧？)注意，这意味着在布尔运算符之后的所有代码都不会执行。

这有什么用呢？它主要是避免了无用地执行代码，可以作为一种技巧使用，假设用户应该输入他/她的名字，但也可以选择什么都不输入，这时可以使用默认值 `"<unknown>"`。可以使用 `if` 语句，但是可以很简洁的方式：

```
name = raw_input("Please enter your name: ") or "<unknown>"
```

换句话说，如果 `raw_input(...)` 语句的返回值为真(不是空字符串)，那么它的值就会赋值给 `name` ，否则将默认的 `"<unknown>"` 赋值给`name``。

这类短路逻辑可以用来实现C和Java中所谓的三元运算符(或条件运算符)。在Python2.5中有一个内置的条件表达式，像下面这样：

```
a if b else c
```

如果b为真，返回a，否则，返回c。(注意，这个运算符不用引入临时变量，就可以直接使用，从而得到与 `raw_input(...)` 例子中同样的结果)

5.4.7 断言

`if` 语句有个非常有用的“近亲”，它的工作方式多少有点像下面这样(伪代码)：

```
if not condition:
    crash program
```

究竟为什么会需要这样的代码呢？就是因为与其让程序在晚些时候崩溃，不如在错误条件出现时直接让它崩溃。一般来说，你可以要求某些条件必须为真(例如，在检查函数参数的属性时，或者作为初期测试和调试过程中的辅助条件)。语句中使用的关键字是 `assert` 。

```
>>> age = 10
>>> assert 0 < age < 100
>>> age = -1
>>> assert 0 < age < 100
    Traceback (most recent call last):
    File "<stdin>", line 1, in <module> AssertionError
```

如果需要确保程序中的某一个条件一定为真才能让程序正常工作的话，assert语句就有用了，他可以在程序中置入检查点。

条件后可以添加字符串，用来解释断言：

```
>>> age = -1
>>> assert 0 < age < 100, "The age must be realistic"
    Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    AssertionError: The age must be realistic
```

5.5 循环

现在你已经知道当条件为真(或假)时如何执行了，但是怎么才能重复执行多次呢？例如，需要实现一个每月提醒你付房租的程序，但是就我们目前学习到的知识而言，需要向下面这样编写程序(伪代码)：

```
发邮件
等一个月
发邮件
等一个月
发邮件
等一个月
(继续下去······)
```

但是如果想让程序继续执行直到认为停止它呢？比如想像下面这样做(还是伪代码)：

```
当我们没有停止时：
    发邮件
    等一个月
```

或者换个简单些的例子。假设想要打印1~100的所有数字，就得再次用这个笨方法：

```
print 1
print 2
print 3
······
print 99
print 100
```

但是如果准备用这种笨方法也就不会学Python了，对吧？

5.5.1 `while` 循环

为了避免上例中笨重的代码，可以像下面这样做：

```
x = 1
while x <= 100
    print x
    x += 1
```

那么Python里面应该如何写呢？你猜对了，就像上面那样。不是很复杂吧？一个循环就可以确保用户输入了名字：

```
name = ""
while not name:
    name = raw_input("Please enter your name: ")
    print "Hello, %s!" % name
```

运行这个程序看看，然后在程序要求输入名字时按下回车键。程序会再次要求输入名字，因为 `name` 还是空字符串，其求值结果为 `False` 。

注：如果直接输入一个空格作为名字又会如何？试试看。程序会接受这个名字，因为包括一个空格的字符串并不是空的，所以不会判定为假。小程序因此出现了瑕疵，修改起来也很简单：只需要把 *`while not name`* 改为 *`while not name or name.isspace()`* 即可，或者可以使用 *`while not name.strip()`* 。

5.2.2 `for` 循环

`while` 语句非常灵活。它可以用来在任何条件为真的情况下重复执行一个代码块。一般情况下这样用就够了，但是有些时候还得量体裁衣。比如要为一个集合(序列和其他可迭代对象)的每个元素都执行一个代码块。

注：可迭代对象是指可以按次序迭代的对象(也就是用于 *`for`* 循环中的)。有关可迭代和迭代器的更多信息，请参见第九章，现在读者可以将其看做序列。

这个时候可以使用 `for` 语句：

```
words = ["this", "is", "an", "ex", "parrot"]
for word in words:
    print word
# 或者
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
for number in numbers:
    print number
# 因为迭代(循环的另一种说法)某范围的数字是很常见的，所以有个内建的范围函数提供使用:
>>> range(0, 10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
# Range函数的工作方式类似于分片。它包含下限(本例中为0)，但不包含上限(本例中为10)。如果希望下限为0，
可以只提供上限:
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
# 下面的程序会打印1~100的数字:
for number in range(1, 100):
    print number
# 它比之前的while循环更简洁。
```

注：如果能使用 *`for`* 循环，就尽量不用 *`while`* 循环。

`xrange` 函数的循环行为类似于 `range` 函数，区别在于 `range` 函数一次创建整个序列，而 `xrange` 一次只创建一个数(在Python3.0中， `range` 会被转换成 `xrange` 风格的函数)。当需要迭代一个巨大的序列时 `xrange` 会更高效，不过一般情况下不需要过多关注它。

5.5.3 循环遍历字典元素

一个简单的 `for` 语句就能遍历字典的所有键，就像遍历访问序列一样：

```
d = {"x": 1, "y": 2, "z": 3}
for key in d:
    print key, "corresponds to", d[key]
```

在Python2.2之前，还只能用 `keys` 等字典方法来获取键(因为不允许直接迭代字典)。如果只需要值，可以使用 `d.values` 替代 `d.keys` 。 `d.items` 方法会将键-值对作为元组返回， `for` 循环的一大好处就是可以循环中使用序列解包：

```
for key, value in d.items():
    print key, "corrsponds", value
```

注：字典元素的顺序通常是没有定义的。换句话说，迭代的时候，字典中的键和值都能保证被处理，但是处理顺序不确定。如果顺序很重要的话，可以将键值保存在单独的列表中，例如在迭代前进行排序。

5.5.4 一些迭代工具

在Python中迭代序列(或者其他可迭代对象)时，有一些函数非常好用。有些函数位于`itertools`模块中(第十章中介绍)，还有一些Python的内建函数也十分方便。

1. 并行迭代

程序可以同时迭代两个系列。比如有下面两个列表：

```
names = ["XuHoo", "Marlowes", "GuoYing", "LeiLa"]
ages = [19, 19, 22, 22]
# 如果想要打印名字和对应的年龄，可以像下面这样做:
for i in range(len(names)):
    print names[i], "is", ages[i], "years old"
```

这里 `i` 是循环索引的标准变量名(可以自己随便定义，一般情况下`for`循环都以 `i` 作为变量名)。

而内建的`zip`函数就可以用来进行并行迭代，可以把两个序列“压缩”在一起，然后返回一个元组的列表：

```
>>> zip(names, ages)
[("XuHoo", 19), ("Marlowes", 19), ("GuoYing", 22), ("LeiLa", 22)]
# 现在我可以在循环中解包元组:
for name, age in zip(names, ages):
    print name, "is", age, "years old"

# zip函数也可以作用于任意多的序列。关于它很重要的一点是zip可以处理不等长的序列，当最短的序列"用完"的时候就会停止:
>>> zip(range(5), xrange(100000000))
[(0, 0), (1, 1), (2, 2), (3, 3), (4, 4)]
```

在上面的代码中，不推荐用`range`替换`xrange`——尽管只需要前五个数字，但`range`会计算所有的数字，这要花费很长的时间。而是用`xrange`就没这个问题了，它只计算前五个数字。

2.按索引迭代

有些时候想要迭代访问序列中的对象，同时还要获取当前对象的索引。例如，在一个字符串列表中替换所有包含`"xxx"`的子字符。实现的方法肯定有很多，假设你想像下面这样做：

```
for string in strings: if "xxx" in string:
        index = strings.index(string)
        # Search for the string in the list of strings
        strings[index] = "[censored]"

# 没问题，但是在替换前要搜索给定的字符串似乎没必要。如果不替换的话，搜索还会返回错误的索引(前面出现的同一个词的索引)。一个比较好的版本如下:
index = 0 for string in strings:
    if "xxx" in string:
        strings[index] = "[censored]"
        index += 1
```

方法有些笨，不过可以接受。另一种方法是使用内建的 `enumerate` 函数：

```
for index, string in enumerate(strings):
    if "xxx" in string:
        strings[index] = "[censored]"
```

这个函数可以在提供索引的地方迭代索引-值对。

3. 翻转和排序迭代

让我们看看另外两个有用的函数： `reversed` 和 `sorted` 。它们同列表的 `reverse` 和 `sort` (`sorted` 和 `sort` 使用同样的参数)方法类似，但作用于任何序列或可迭代对象上，不是原地修改对象，而是返回翻转或排序后的版本：

```
>>> sorted([4, 3, 6, 8, 3])
[3, 3, 4, 6, 8]
>>> sorted("Hello, world!")
[' ', '!', ',', 'H', 'd', 'e', 'l', 'l', 'l', 'o', 'o', 'r', 'w']
>>> list(reversed("Hello, world!"))
['!', 'd', 'l', 'r', 'o', 'w', ' ', ',', 'o', 'l', 'l', 'e', 'H']
>>> "".join(reversed("Hello, world!"))
'!dlrow ,olleH'
```

注意，虽然 `sorted` 方法返回列表， `reversed` 方法却返回一个更加不可思议的可迭代对象。它们具体的含义不用过多关注，大可在 `for` 循环以及 `join` 方法中使用，而不会有任何问题。不过却不能直接对它使用索引、分片以及调用 `list` 方法，如果希望进行上述处理，那么可以使用 `list` 类型转换返回对象，上面的例子中已经给出具体的做法。

5.5.5 跳出循环

一般来说，循环会一直执行到条件为假，或者到序列元素用完时。但是有些时候可能会提前中断一个循环，进行新的迭代(新一"轮"的代码执行)，或者仅仅就是像结束循环。

1. `break`

结束(跳出)循环可以使用 `break` 语句。假设需要寻找100以内的最大平方数，那么程序可以开始从100往下迭代到0.当找到一个平方数时就不需要继续循环了，所以可以跳出循环：

```
from math import sqrt
    for n in range(99, 0, -1):
        root = sqrt(n)
        if root == int(root):
            print n break
```

如果执行这个程序的话，会打印出 `81` ，然后程序停止。注意，上面的代码中 `range` 函数增加了第三个参数——表示步长，步长表示每对相邻数字之间的差别。将其设置为负值的话就会想例子中一样反向迭代。它也可以用来跳过数字：

```
>>> range(0, 10, 2)
[0, 2, 4, 6, 8]
```

2. `continue`

`continue` 语句比 `break` 语句用得要少得多。它会让当前的迭代结束，“跳”到下一轮循环的开始。它最基本的意思是“跳过剩余的循环体，但是不结束循环”。当循环体很大而且很复杂的时候，这会很有用，有些时候因为一些原因可能会跳过它——这个时候可以使用 `continue` 语句：

```
for x in seq:
    if condition1:
    continue
    if condition2:
    continue
    if condition3:
    continue
    do_something()
    do_something_else()
    do_another_thing()
    etc()
# 很多时候，只要使用if语句就可以了:
for x in seq:
    if not (condition1 or condition2 or condition3):
        do_something()
        do_something_else()
        do_another_thing()
        etc()
```

尽管 `continue` 语句非常有用，它却不是最本质的。应该习惯使用 `break` 语句，因为在 `while True` 语句中会经常用到它。下一节会对此进行介绍。

3. `while True/break` 习语

Python中的 `while` 和 `for` 循环非常灵活，但一旦使用 `while` 语句就会遇到一个需要更多功能的问题。如果需要当用户在提示符下输入单词时做一些事情，并且在用户不输入单词后结束循环。可以使用下面的方法：

```
word = "dummy"
while word:
    # 处理word
    word = raw_input("Please enter a word: ")
    print "The word was " + word
# 下面是一个会话示例:
Please enter a word: first
The word was first
Please enter a word: second
The word was second
Please enter a word:
```

代码按要求的方式工作(大概还能做些比直接打印出单词更有用的工作)。但是代码有些丑。在进入循环之前需要给word赋一个哑值(未使用的)。使用哑值(dummy value)就是工作没有尽善尽美的标志。让我们试着避免它：

```
word = raw_input("Please enter a word: ")
# 处理word
while word:
    print "The word was " + word
    word = raw_input("Please enter a word: ")
    # 哑值没有了。但是有重复的代码(这样也不好):要用一样的赋值语句在两个地方两次调用raw_input。能否不
这么做呢？可以使用while True/break语句:
while True:
    word = raw_input("Please enter a word: ")
    if not word:
    break
    # 处理word
    print "The word was " + word
```

`while True` 的部分实现了一个永远不会自己停止的循环。但是在循环内部的if语句中加入条件也是可以的，在条件满足时使用 `break` 语句。这样一来就可以在循环内部任何地方而不是只在开头(像普通的 `while` 循环一样)终止循环。 `if/break` 语句自然地将循环分为两部分：第一部分负责初始化(在普通的 `while` 循环中，这部分需要重复)，第二部分则在循环条件为真的情况下使用第一部分内初始化好的数据。

尽管应该避免在代码中频繁使用 `break` 语句(因为这可能会让循环的可读性降低，尤其是在一个循环中使用多个 `break` 语句的时候)，但这个特殊的技术用得非常普遍，大多数Python程序员(包括你自己)都能理解你的意思。

5.5.6 循环中的 `else` 子句

当在循环内使用 `break` 语句时，通常是因为“找到”了某物或者因为某事“发生”了。在跳出是做一些事情是很简单的(比如 `print n`)，但是有些时候想要在没有跳出之前做些事情。那么怎么判断呢？可以使用布尔变量，再循环前将其设定为 `False` ，跳出后设定为 `True` 。然后再使用 `if` 语句查看循环是否跳出了：

```
broke_out = False for x in seq:
    do_something(x)
    if condition(x):
        broke_out = True
        break
    do_something_else(x)
    if not broke_out:
        print "I didn't break out!"

# 更简单的方式是在循环中增加一个else子句——它仅在没有调用`break`时执行。让我们用这个方法重写刚才的例子:
from math import sqrt
for n in range(99, 81, -1):
    root = sqrt(n)
    if root == int(root):
        print n break
else:
    print "Didn't find it!"
```

注意我将下限改为81(不包括81)以测试 `else` 子句。如果执行程序的话，它会打印出 `"Didn't find it!"` ，因为(就像在 `break` 那节看到的一样)100以内最大的平方数时81。 `for` 和 `while` 循环中都可以使用 `continue` 、 `break` 语句和 `else` 子句。

5.6 列表推导式——轻量级循环

列表推导式(list comprehension)是利用其他列表创建新列表(类似于数学术语中的集合推导式)的一种方法。它的工作方式类似于 `for` 循环，也很简单：

```
>>> [x*x for x in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

列表有 `range(10)` 中每个x的平方组成。太容易了？如果只想打印出那些能被3整除的平方数呢？那么可以使用模除运算符—— `y%3` ，当数字可以被3整除时返回0(注意， `x` 能被3整除时， `x` 的平方必然也可以被3整除)。这个语句可以通过增加一个 `if` 部分添加到列表推导式中：

```
>>> [x*x for x in range(10) if x % 3 == 0]
[0, 9, 36, 81]
# 也可以增加更多for语句的部分:
>>> [(x, y) for x in range(3) for y in range(3)]
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2)]
# 作为对比，下面的代码使用两个for语句创建了相同的列表:
result = []
for x in range(3):
    for y in range(3):
        result.append((x, y))
print result
# 也可以和if子句联合使用，像以前一样:
>>> girls = ["alice", "bernice", "clarice"] >>> boys = ["chris", "arnold", "bob"]
>>> [b+"+"+g for b in boys for g in girls if b[0] == g[0]]
['chris+clarice', 'arnold+alice', 'bob+bernice'] # 这样就得到了那些名字首字母相同的男孩和女孩。
```

注：使用普通的圆括号而不是方括号不会得到“元组推导式”。在*Python2.3*及以前的版本中只会得到错误。在最近的版本中，则会得到一个生成器。请参见*9.7*节获得更多信息。

更优秀的方案

男孩/女孩名字对的例子其实效率不高，因为它会检查每个可能的配对。Python有很多解决这个问题的方法，下面的方法是Alex Martelli推荐的：

```
girls = ["alice", "bernice", "clarice"]
boys = ["chris", "arnold", "bob"]
letterGirls = {}
for girl in girls:
    letterGirls.setdefault(girl[0], []).append(girl)
print [b+"+"+g for b in boys for g in letterGirls[b[0]]]
```

这个程序创建了一个叫做 `letterGirls` 的字典，其中每一项都把单字母作为键，以女孩名字组成的列表作为值。(`setdefault` 字典方法在前一章中已经介绍过)在字典建立后，列表推导式循环整个男孩集合，并且查找那些和当前男孩名字首字母相同的女孩集合。这样列表推导式就不用尝试所有的男孩女孩的组合，检查首字母是否匹配。

5.7 三人行

作为本章的结束，让我们走马观花地看一下另外三个语句： `pass` 、 `del` 和 `exec` 。

5.7.1 什么都没发生

有的时候，程序什么事情都不用做吗。这种情况不多，但是一旦出现，就应该让 `pass` 语句出马了。

```
>>> pass
>>>
```

似乎没什么动静。

那么究竟为什么使用一个什么都不做的语句？它可以在代码中做占位符使用。比如程序需要一个 `if` 语句，然后进行测试，但是缺少其中一个语句块的代码，考虑下面的情况：

```
if name == "Ralph Auldus Melish":
    print "Welcome!"
elif name == "End":
    # 还没完……
elif name == "Bill Gates":
    print "Access Denied"

# 代码不会执行，因为Python中空代码块是非法的。解决方案就是在语句块中加上一个pass语句：
if name == "Ralph Auldus Melish":
    print "Welcome!"
elif name == "End":
    # 还没完……
    pass
elif name == "Bill Gates":
    print "Access Denied"
```

注：注释和 `pass` 语句联合的代替方案是插入字符串。对于那些没有完成的函数*(*参见第六章*)*和类*(*参见第七章*)*来说这个方法尤其有用，因为它们会扮演文档字符串*(docstring)*的角色*(*第六章中会有解释*)*。

5.7.2 使用del删除

一般来说，Python会删除那些不再使用的对象(因为使用者不会再通过任何变量或数据结构引用它们)：

```
>>> scoundrel = {"age": 42, "first name": "Robin", "last name": "of Locksley"} >>> rob
in = scoundrel >>> scoundrel
{'last name': 'of Locksley', 'first name': 'Robin', 'age': 42}
>>> robin
{'last name': 'of Locksley', 'first name': 'Robin', 'age': 42}
>>> scoundrel = None >>> robin = None
```

首先， `robin` 和 `scoundrel` 都被绑定到同一个字典上。所以当设置 `scoundrel` 为 `None` 的时候，字典通过 `robin` 还是可用的。但是当我把 `robin` 也设置为 `None` 的时候，字典就“漂”在内存里了，没有任何名字绑定到它上面。没有办法获取和使用它，所以Python解释器(以其无穷的智慧)直接删除了那个字典(这种行为被称为垃圾收集)。注意，也可以使用 `None` 之外的其他值。字典同样会“消失不见”。

另外一个方法就是使用 `del` 语句(我们在第二章和第四章里面用来删除序列和字典元素的语句)，它不仅会移除一个对象的引用，也会移除那个名字本身。

```
>>> x = 1
>>> del x
>>> x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  NameError: name 'x' is not defined
# 看起来很简单，但有时理解起来有些难度。例如，下面的例子中，x和y都指向同一个列表:
>>> x = ["Hello", "world"]
>>> y = x
>>> y[1] = "Python"
>>> x
['Hello', 'Python']
# 会有人认为删除x后，y也就随之消失了，但并非如此:
>>> del x
>>> y
['Hello', 'Python']
```

为什么会这样？`x` 和 `y` 都指向同一个列表。但是删除 `x` 并不会影响 `y` 。原因就是删除的只是名称，而不是列表本身(值)。事实上，在Python中是没有办法删除值的(也不需要过多考虑删除值的问题，因为在某个值不再使用的时候，Python解释器会负责内存的回收)。

5.7.3 使用 `exec` 和 `eval` 执行和求值字符串

有些时候可能会需要动态地创造Python代码，然后将其作为语句执行或作为表达式计算，这可能近似于“黑暗魔法”——在此之前，一定要慎之又慎，仔细考虑。

警告：本节中，会学到如何执行存储在字符串中的*Python*代码。这样做会有很严重的潜在安全漏洞。如果程序将用户提供的一段内容中的一部分字符串作为代码执行，程序可能会失去对代码执行的控制，这种情况在网络应用程序——比如*CGI*脚本中尤其危险，这部分内容会在第十五章介绍。

1. `exec`

执行一个字符串的语句是 `exec` (在Python3.0中， `exec` 是一个函数而不是语句)：

```
>>> exec "print 'Hello, world!'"
Hello, world!
```

但是，使用简单形式的 `exec` 语句绝不是好事。很多情况下可以给它提供命名空间——可以放置变量的地方。你想这样做，从而使代码不会干扰命名空间(也就是改变你的变量)，比如，下面的代码中使用了名称 `sqrt` ：

```
>>> from math import sqrt
>>> exec "sqrt = 1"
>>> sqrt(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  TypeError: 'int' object is not callable
```

想想看，为什么一开始我们要这样做？ `exec` 语句最有用的地方在于可以动态地创建代码字符串。如果字符串是从其他地方获得——很有可能是用户——那么几乎不能确定其中到底包含什么代码。所以为了安全起见，可以增加一个字典，起到命名空间的作用。

注：命名空间的概念，或称为作用域(*scope*)，是非常重要的知识，下一章会深入学习，但是现在可以把它想象成保存变量的地方，类似于不可见的字典。所以在程序执行 *x=1* 这类赋值语句时，就将键 *x* 和值 *1* 放在当前的命名空间内，这个命名空间一般来说都是全局命名空间(到目前为止绝大多数都是如此)，但这并不是必须的。

可以通过增加 `in<scope>` 来实现，其中 `<scope>` 就是起到放置代码字符串命名空间作用的字典。

```
>>> from math import sqrt
>>> scope = {}
>>> exec "sqrt = 1" in scope
>>> sqrt(4) 2.0
>>> scope["sqrt"]
1
```

可以看到，潜在的破坏性代码并不会覆盖 `sqrt` 函数，原来的函数能正常工作，而通过 `exec` 赋值的变量 `sqrt` 只在它的作用域内有效。

注意，如果需要将 `scope` 打印出来的话，会看到其中包含很多东西，因为内建的 `__builtins__` 字典自动包含所有的内建函数和值：

```
>>> len(scope) 2
>>> scope.keys()
['__builtins__', 'sqrt']
```

2. `eval`

`eval` (用于“求值”)是类似于 `exec` 的内建函数。 `exec` 语句会执行一系列*Python*语句，而 `eval` 会计算*Python*表达式(以字符串形式书写)，并且返回结果值。(`exec` 语句并不返回任何对象，因为它本身就是语句)例如，可以使用下面的代码创建一个Python计算器：

```
>>> eval(raw_input("Enter an arithmetic expression: "))
Enter an arithmetic expression: 6 + 18 * 2
42
```

注：表达式 *eval(raw_input(...))* 事实上等同于 *input(...)* 。在*Python3.0*中， *raw_input* 被重命名为 *input* 。

跟 `exec` 一样， `eval` 也可以使用命名空间。尽管表达式几乎不像语句那样为变量重新赋值。(实际上，可以给 `eval` 语句提供两个命名空间，一个全局的一个局部的。全局的必须是字典，局部的可以是任何形式的映射。)

警告：尽管表达式一般不给变量重新赋值，但它们的确可以(比如可以调用函数给全局变量重新赋值)。所以使用 `eval` 语句对付一些不可信任的代码并不比 `exec` 语句安全。目前，在Python内没有任何执行不可信任代码的安全方式。一个可选的方案是使用Python的实现，比如Jython(参见第十七章)，以及使用一些本地机制，比如Java的sandbox功能。

初探作用域

给 `exec` 或者 `eval` 语句提供命名空间时，还可以在真正使用命名空间前放置一些值进去：

```
>>> scope = {}
>>> scope["x"] = 2
>>> scope["y"] = 3
>>> eval("x * y", scope)
6

# 同理，exec或者eval调用的作用域也能在另一个上面使用:
>>> scope = {}
>>> exec "x = 2" in scope
>>> eval("x * x", scope)
4
```

事实上， `exec` 语句和 `eval` 语句并不常用，但是它们可以作为后兜里的得力工具(当然，这仅仅是比喻而已)。

5.8 小结

本章中介绍了几类语句和其他知识。

√打印。 `print` 语句可以用来打印由逗号隔开的多个值。如果语句以逗号结尾，后面的 `print` 语句会在同一行内继续打印。

√导入。有些时候，你不喜欢你想导入的函数名——还有可能由于其他原因使用了这个函数名。可以使用 `from ... as ...` 语句进行函数的局部重命名。

√赋值。通过序列解包和链式赋值功能，多个变量赋值可以一次性赋值，通过增量赋值可以原地改变变量。

√块。块是通过缩排使语句成组的一种方法。它们可以在条件以及循环语句中使用，后面的章节中会介绍，块也可以在函数和类中使用。

√条件。条件语句可以根据条件(布尔表达式)执行或不执行一个语句块。几个条件可以串联使用 `if/elif/else` 。这个主题下还有一种变体叫做条件表达式，形如 `a if b else c` (这种表达式其实类似于三元运算)。

√断言。断言简单来说就是肯定某事(布尔表达式)为真。也可在后面跟上这么认为的原因。如果表达式为真，断言就会让程序崩溃(事实上是产生异常——第八章会介绍)。比起错误潜藏在程序中，直到你不知道它源在何处，更好的方法是尽早找到错误。

√ 循环。可以为序列(比如一个范围内的数字)中的每一个元素执行一个语句块，或者在条件为真的时候继续执行一段语句。可以使用 `continue` 语句跳过块中的其他语句，然后继续下一次迭代，或者使用 `break` 语句跳出循环。还可以选择在循环结尾加上 `else` 子句，当没有执行循环内部的 `break` 语句的时候便会执行 `else` 子句中的内容。

√ 列表推导式。它不是真正的语句，而是看起来像循环的表达式，这也是我将它归到循环语句中的原因。通过列表推导式，可以从旧列表中产生新的列表、对元素应用函数、过滤不需要的元素，等等。这个功能很强大，但是很多情况下，直接使用循环和条件语句(工作也能完成)，程序会更易读。

√ `pass` 、 `del` 、 `exec` 和 `eval` 语句。 `pass` 语句什么都不做，可以作为占位符使用。 `del` 语句用来删除变量，或者数据结构的一部分，但是不能用来删除值。 `exec` 语句用于执行Python程序相同的方式来执行字符串。内建的 `eval` 函数对写在字符串中的表达式进行计算并且返回结果。

5.8.1 本章的新函数

本章涉及的新函数如表5-2所示。

表5-2 本章的新函数

```
chr(n)                                     当传入序号n时，返回n所代表的包含一个字符的字符串(0≤n&lt
;256)。
eval(source[, globals[, locals]])          将字符串作为表达式计算，并且返回值。
enumerate(seq)                             产生用于迭代的(索引，值)对。
ord(c)                                     返回单字符字符串的int值。
range([start,] stop[, step])               创建整数的列表。
reversed(seq)                              产生seq中值的反向版本，用于迭代。
sorted(seq[, cmp][, key][, reverse])       返回seq中值排序后的列表。
xrange([start,] stop[, step])              创造xrange对象用于迭代。
zip(seq1, seq2 ...)                        创造用于并行迭代的新序列。
```

5.8.2 接下来学什么

现在基本知识已经学完了。实现任何自己能想到的算法已经没问题了，也可以让程序读取参数并且打印结果。下面两章中，将会介绍可以创建较大程序，却不让代码冗长的知识。这也就是我们所说的抽象(abstraction)。

第六章 抽象

> 来源：http://www.cnblogs.com/Marlowes/p/5351415.html
>
> 作者：Marlowes

本章将会介绍如何将语句组织成函数，这样，你可以告诉计算机如何做事，并且只需要告诉一次。有了函数以后，就不必反反复复像计算机传递同样的具体指令了。本章还会详细介绍参数(parameter)和作用域(scope)的概念，以及递归的概念及其在程序中的用途。

6.1 懒惰即美德

目前为止我们缩写的程序都很小，如果想要编写大型程序，很快就会遇到麻烦。考虑一下如果在一个地方编写了一段代码，但在另一个地方也要用到这段代码，这时会发生什么。例如，假设我们编写了一小段代码来计算斐波那契数列(任一个数都是前两数之和的数字序列)：

```
fibs = [0, 1] for i in range(8):
    fibs.append(fibs[-2] + fibs[-1])
    # 运行之后,fibs会包含斐波那契数列的前10个数字:
fibs
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
# 如果想要以此计算前10个数的话，没有问题。你甚至可以将用户输入的数字作为动态范围的长度使用，从而改变for语句循环的次数:
fibs = [0, 1]
num = input("How many Fibonacci numbers do you want? ")
for i in range(num - 2):
    fibs.append(fibs[-2] + fibs[-1])
    print fibs
```

注：在本例中，读取字符串可以使用 `raw_input` 函数，然后再用 `int` 函数将其转换为整数。

但是如果想用这些数字做其他事情呢？当然可以在需要的时候重写同样的循环，但是如果已经编写的是一段复杂的代码——比如下载一系列网页并且计算词频——应该怎么做呢？你是否希望在每次需要的时候把所有的代码重写一遍呢？当然不用，真正的程序员不会这么做的，他们都很懒，但不是用错误的方式犯懒，换句话说就是他们不做无用功。

那么真正的程序员怎么做呢？他们会让自己的程序抽象一些。上面的程序可以改写为比较抽象的版本：

```
num = input("How many numbers do you want? ")
print fibs(num)
```

这个程序的具体细节已经写的很清楚了(读入数值，然后打印结果)。事实上计算菲波那切数列是由一种更抽象的方式完成的：只需要告诉计算机去做就好，不用特别说明应该怎么做。名为fibs的函数被创建，然后在需要计算菲波那切数列的地方调用它即可。如果这函数要被调用

很多次的话，这么做会节省很多精力。

6.2 抽象和结构

抽象可以节省很多工作，实际上它的作用还要更大，它是使得计算机程序可以让人读懂的关键(这也是最基本的要求，不管是读还是写程序)。计算机非常乐于处理精确和具体的指令，但是人可就不同了。如果有人问我去电影院怎么走，估计他不会希望我回答“向前走10步，左转90度，再走5步右转45度，走123步”。弄不好就迷路了，对吧？

现在，如果我告诉他“一直沿着街走，过桥，电影院就在左手边”，这样就明白多了吧！关键在于大家都知道怎么走路和过桥，不需要明确指令来指导这些事。

组织计算机程序也是类似的。程序应该是非常抽象的，就像“下载网页、计算频率、打印每个单词的频率”一样易懂。事实上，我们现在就能把这段描述翻译成Python程序：

```
page = download_page()
freqs = compute_frequencies(page)
for word, freq in freqs:
    print word, freq
```

虽然没有明确地说出它是怎么做的，单读完代码就知道程序做什么了。只需要告诉计算机下载网页并计算词频。这些操作的具体指令细节会在其他地方给出——在单独的函数定义中。

6.3 创建函数

函数是可以调用的(可能带有参数，也就是放在圆括号中的值)，它执行某种行为并且返回一个值(并非所有Python函数都有返回值)。一般来说，内建的 `callable` 函数可以用来判断函数是否可调用：

```
>>> import math >>> x = 1
>>> y = math.sqrt
>>> callable(x)
False
>>> callable(y)
True
```

注：函数 *`callable`* 在*Python3.0*中不再可用，需要使用表达式 *`hasattr(func, __call__)`* 代替，有关 *`hasattr`* 的更多信息，请参见第七章。

就像前一节内容中介绍的，创建函数是组织程序的关键。那么怎么定义函数呢？使用 `def` (或“函数定义”)语句即可：

```
def hello(name):
    return "Hello, " + name + "!"

# 运行这段程序就会得到一个名为hello的新函数，它可以返回一个将输入的参数作为名字的问候语。可以像使用内建函数一样使用它:
>>> print hello("world")
Hello, world!
>>> print hello("XuHoo")
Hello, XuHoo!
```

很精巧吧？那么想想看怎么写个返回斐波那契数列列表的函数吧。简单！只需要使用刚才的代码，把从用户输入获取的数字改为作为参数接收数字：

```
num = input("How many numbers do you want? ")
def fibs(num):
    result = [0, 1]
    for i in range(num - 2):
        result.append(result[-2] + resultp[-1])
    return result
# 执行这段与语句后，编译器就知道如何计算斐波那契数列了——所以现在就不用关注细节了，只要用函数fibs就行:
>>> fibs(10)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
>>> fibs(15)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377]
```

本例中的 `num` 和 `result` 的名字都是随便起的，但是 `return` 语句非常重要。 `return` 语句是用来从函数中返回值的(函数可以返回一个以上的值，元组中返回即可)(前例中的 `hello` 函数也有用到)。

6.3.1 文档化函数

如果想要给函数写文档，让其他使用该函数的人能理解的话，可以加入注释(以 `#` 开头)。另外一个方式就是直接写上字符串。这类字符串在其他地方可能会非常有用，比如在 `def` 语句后面(以及在模块或者类的开头——有关类的更多内容请参见第七章，有关模块的更多内容请参见第十章)。如果在函数的开头写下字符串，它就会作为函数的一部分进行存储，这成为文档字符串。下面代码演示了如何给函数添加文档字符串：

```
def square(x):
    "Calculates the square of the number x."
    return x * x
# 文档字符串可以按如下方式访问:
>>> square.__doc__
"Calculates the square of the number x."
```

注： `__doc__` 是函数属性，第七章中会介绍更多关于属性的知识，属性名中的双下划线表示它是个特殊属性。这类特殊和“魔法”属性会在第九章讨论。

内建的 `help` 函数是非常有用的。在交互式解释器中使用它，就可以得到关于函数，包括它的文档字符串的信息：

```
>>> help(square)
Help on function square in module __main__;

square(x)
    Calculates the square of the number x.
```

第十章中会再次对`help`函数进行讨论。

6.3.2 并非真正函数的函数

数学意义上的函数，总在计算其参数后返回点什么。Python的有些函数却并不返回任何东西。在其他语言中(比如Pascal)，这类函数可能有其他名字(比如过程)。但是Python的函数就是函数，即便它从学术上讲并不是函数。没有`return`语句，或者虽有`return`语句但`return`后边没有跟任何值的函数不返回值：

```
def test():
    print "This is printed"
    return
    print "This is not"

# 这里的return语句只起到结束函数的作用:
>>> x = test()
This is printed
# 可以看到，第2个print语句被跳过了(类似于循环中的break语句，不过这里是跳出函数)。但是如果test不返回任何值，那么x又引用什么呢？让我们看看:
>>> x
>>>
# 没东西，再仔细看看:
>>> print x
>>> None
```

好熟悉的值：`None`。所以所有的函数的确都返回了东西：当不需要它们返回值的时候，它们就返回`None`。看来刚才“有些函数并不真的是函数”的说法有些不公平了。

注：千万不要被默认行为所迷惑。如果在*`if`*语句内返回值，那么要确保其他分支也有返回值，这样一来当调用者期待一个序列的时候，就不会意外地返回*`None`*。

6.4 参数魔法

函数使用起来很简单，创建起来也不复杂。但函数参数的用法有时就有些神奇了。还是先从最基础的介绍起。

6.4.1 值从哪里来

函数被定义后，所操作的值是从哪里来的呢？一般来说不用担心这些，编写函数只是给程序需要的部分(也可能是其他程序)提供服务，能保证函数在被提供给可接受参数的时候正常工作就行，参数错误的话显然会导致失败(一般来说这时候要用断言和异常，第八章会介绍异常)。

注：写在`def`语句中函数名后面的变量通常叫做函数的形参，而调用函数的时候提供的值是实参，或者称为参数。一般来说，本书在介绍的时候对于两者的区别并不会吹毛求疵。如果这种区别影响较大的话，我会将实参称为“值”以区别与形参。

6.4.2 我能改变参数吗

函数通过它的参数获得一系列值。那么这些值能改变吗？如果改变了又会怎么样？参数只是变量而已，所以它们的行为其实和你预想的一样。在函数内为参数赋予新值不会改变外部任何变量的值：

```
>>> def try_to_change(n):
...     n = "Mr. XuHoo"
...
>>> name = "Mr. Marlowes"
>>> try_to_change(name)
>>> name
'Mr. Marlowes'

# 在try_to_change内，参数n获得了新值，但是它没有影响到name变量。n实际上是个完全不同的变量，具体的工作方式类似于下面这样:
>>> name = "Mr. Marlowes"
>>> n = name
# 这句的作用基本上等于传参数
>>> n = "Mr. XuHoo"
# 在函数内部完成的
>>> name
'Mr. Marlowes'
```

结果是显而易见的。当变量`n`改变的时候，变量`name`不变。同样，当在函数内部把参数重绑(赋值)的时候，函数外的变量是不会受到影响的。

注：参数存储在局部作用域*(local scope)*内，本章后面会介绍。

字符串(以及数字和元组)是不可变的，即无法被修改(也就是说只能用新的值覆盖)。所以它们做参数的时候也就无需多做介绍。但是考虑一下如果将可变的数据结构如列表用作参数的时候会发生什么：

```
>>> def change(n):
...     n[0] = "Mr. XuHoo"
...
>>> names = ["Mrs. Marlowes", "Mrs. Something"]
>>> change(names)
>>> names
['Mr. XuHoo', 'Mrs. Something']
```

本例中，参数被改变了。这就是本例和前面例子中至关重要的区别。前面的例子中，局部变量被赋予了新值，但是这个例子中变量`names`所绑定的列表的确变了。有些奇怪吧？其实这种行为并不奇怪，下面不用函数调用再做一次：

```
>>> names = ["Mrs. Marlowes", "Mrs. Something"]
>>> n = names  # 再来一次，模拟传参行为
>>> n[0] = "Mr. XuHoo"  # 改变列表
>>> names
['Mr. XuHoo', 'Mrs. Something']
```

这类情况在前面已经出现了多次。当两个变量同时引用一个列表的时候，它们的确是同时引用一个列表。就是这么简单。如果想避免出现这种情况，可以复制一个列表的副本。当在序列中做切片的时候，返回的切片总是一个副本。因此，如果你复制了整个列表的切片，将会得到一个副本：

```
>>> names = ["Mrs. Marlowes", "Mrs. Something"]
>>> n = names[:]
# 现在n和names包含两个独立(不同)的列表，其值相等:
>>> n is names
False '
>>> n == names
True
# 如果现在改变n(就像在函数change中做的一样)，则不会影响到names:
>>> n[0] = "Mr. XuHoo"
>>> n
['Mr. XuHoo', 'Mrs. Something']
>>> names
['Mrs. Marlowes', 'Mrs. Something']
# 再用change试一下:
>>> change(names[:]) >>> names
['Mrs. Marlowes', 'Mrs. Something']
```

现在参数 `n` 包含一个副本，而原始的列表是安全的。

注：可能有的读者会发现这样的问题：函数的局部名称——包括参数在内——并不和外面的函数名称(全局的)冲突。关于作用域的更多信息，后面的章节会进行讨论。

1. 为什么要修改参数

使用函数改变数据结构(比如列表或字典)是一种将程序抽象化的好方法。假设需要编写一个存储名字并且能用名字、中间名或姓查找联系人的程序，可以使用下面的数据结构：

```
storage = {}
storage["first"] = {}
storage["middle"] = {}
storage["last"] = {}
```

`storage` 这个数据结构是带有3个键 `“first”` 、 `“middle”` 、 `“last”` 的字典。每个键下面都又存储一个字典。子字典中，可以使用名字(名字、中间名或姓)作为键，插入联系人列表作为值。比如要把我自己的名字加入这个数据结构，可以像下面这么做：

```
>>> me = "Magnus Lie Hetland"
>>> storage["first"]["Magnus"] = [me]
>>> storage["middle"]["Lie"] = [me]
>>> storage["last"]["Hetland"] = [me]
# 每个键下面都存储了一个以人名组成的列表。本例中，列表中只有我。
# 现在如果想要得到所有注册的中间名为Lie的人，可以像下面这么做:
>>> storage["middle"]["Lie"]
['Magnus Lie Hetland']
```

将人名加到列表中的步骤有点枯燥乏味，尤其是要加入很多姓名相同的人时，因为需要扩展已经存储了那些名字的列表。例如，下面加入我姐姐的名字，而且假设不知道数据库中已经存储了什么：

```
>>> my_sister = "Anne Lie Hetland"
>>> storage["first"].setdefault("Anne", []).append(my_sister)
>>> storage["middle"].setdefault("Lie", []).append(my_sister)
>>> storage["last"].setdefault("Hetland", []).append(my_sister)
>>> storage["first"]["Anne"]
['Anne Lie Hetland'] >>> storage["middle"]["Lie"]
['Magnus Lie Hetland', 'Anne Lie Hetland']
```

如果要写个大程序来这样更新列表，那么很显然程序很快就会变得臃肿且笨拙不堪了。

抽象的要点就是隐藏更新时繁琐的细节，这个过程可以用函数实现。下面的例子就是初始化数据结构的函数：

```
def init(data):
    data["first"] = {}
    data["middle"] = {}
    data["last"] = {}
# 上面的代码只是把初始化语句放到了函数中，使用方法如下：
>>> storage = {}
>>> init(storage)
>>> storage
{'middle': {}, 'last': {}, 'first': {}}
```

可以看到，函数包办了初始化的工作，让代码更易读。

注：字典的键并没有特定的顺序，所以当字典打印出来的时候，顺序是不同的。如果读者在自己的解释器中打印出的顺序不同，请不要担心，这是很正常的。

在编写存储名字的函数前，先写个获得名字的函数：

```
def lookup(data, label, name):
    return data[label].get(name)
```

标签(比如 `"middle"`)以及名字(比如 `"Lie"`)可以作为参数提供给 `lookup` 函数使用，这样会获得包含全名的列表。换句话说，如果我的名字已经存储了，可以像下面这样做：

```
>>> lookup(storage, "middle", "Lie")
['Magnus Lie Hetland']
```

注意，返回的列表和存储在数据结构中的列表是相同的，所以如果列表被修改了，那么也会影响数据结构(没有查询到人的时候就问题不大了，因为函数返回的是 `None`)。

```
def store(data, full_name):
    names = full_name.split()
    if len(name) == 2:
        names.insert(1, "")
    labels = "first", "middle", "last"
    for label, name in zip(labels, names):
        people = lookup(data, label, name)
        if people:
            people.append(full_name) else:
            data[label][name] = [full_name]
```

store函数执行以下步骤。

(1) 使用参数 `data` 和 `full_name` 进入函数，这两个参数被设置为函数在外部获得的一些值。

(2) 通过拆分 `full_name` ，得到一个叫做 `names` 的列表。

(3) 如果 `names` 的长度为 `2` (只有首名和末名)，那么插入一个空字符串作为中间名。

(4) 将字符串 `"first"` 、 `"middle"` 和 `"last"` 作为元组存储在 `labels` 中(也可以使用列表，这里只是为了方便而去掉括号)。

(5) 使用 `zip` 函数联合标签和名字，对于每一个 `(label, name)` 对，进行一下处理：

1) 获得属于给定标签和名字的列表；

2) 将 `full_name` 添加到列表中，或者插入一个需要的新列表。

来试用一下刚刚实现的程序：

```
>>> MyNames = {} >>> init(MyNames)
>>> store(MyNames, "Magnus Lie Hetland")
>>> lookup(MyNames, "middle", "Lie")
# 好像可以工作，再试试:
>>> store(MyNames, "Robin Hood")
>>> store(MyNames, "Robin Locksley")
>>> lookup(MyNames, "first", "Robin")
['Robin Hood', 'Robin Locks ley']
>>> store(MyNames, "Mr. XuHoo")
>>> lookup(MyNames, "middle", "")
['Robin Hood', 'Robin Locksley', 'Mr. XuHoo']
```

可以看到，如果某些人的名字、中间名或姓相同，那么结果中会包含所有这些人的信息。

注：这类程序很适合进行面向对象程序设计，下一章内会讨论到如何进行面向对象程序设计。

2.如果我的参数不可变呢

在某些语言(比如C++、Pascal和Ada)中，重新绑定参数并且使这些改变影响到函数外的变量是很平常的事情。但在Python中这是不可能的：函数只能修改参数对象本身。但是如果你的参数不可变(比如是数字)，又该怎么办呢？

不好意思，没有办法。这个时候你应该从函数中返回所有你需要的值(如果值多于一个的话就以元组形式返回)。例如，将变量的数值增1的函数可以这样写：

```
>>> def inc(x):
        return x + 1
...
>>> foo = 10
>>> foo = inc(foo)
>>> foo 11

# 如果真的想改变参数，那么可以使用一点小技巧，即将值放置在列表中:
>>> def inc(x):
        x[0] = x[0] + 1
...
>>> foo = [10]
>>> inc(foo)
>>> foo
[11]
```

这样就会返回新值，代码看起来也比较清晰。

6.4.3 关键字参数和默认值

目前为止我们所使用的参数都叫做位置参数，因为它们的位置很重要，事实上比它们的名字更加重要。本节中引入的这个功能可以回避位置问题，当你慢慢习惯使用这个功能以后，就会发现程序规模越大，它们的作用也就越大。

```
# 考虑下面的两个函数:
def hello_1(greeting, name):
    print "%s, %s!" % (greeting, name)
def hello_2(name, greeting):
    print "%s, %s!" % (name, greeting)
# 两个代码所实现的是完全一样的功能，只是参数顺序反过来了:
>>> hello_1("Hello", "world")
Hello, world!
>>> hello_2("Hello", "world")
Hello, world!
# 有些时候(尤其是参数很多的时候)，参数的顺序是很难记住的。为了让事情简单些，可以提供参数的名字:
>>> hello_1(greeting="Hello", name="world")
Hello, world!
# 这样一来，顺序就完全没影响了:
>>> hello_1(name="world", greeting="Hello")
Hello, world!
# 但参数名和值一定要对应:
>>> hello_2(greeting="Hello", name="world")
world, Hello!
```

这类使用参数名提供的参数叫做关键字参数。它的主要作用在于可以明确每个参数的作用，也就避免了下面这样的奇怪的函数调用：

```
>>> store("Mr. Brainsample", 10, 20, 13, 5)
# 可以使用:
>>> store(patient="Mr. Brainsample", hour=10, minut=20, day=13, month=5)
```

尽管这么做打的字就多了些，但是很显然，每个参数的含义变得更加清晰。而且就算弄乱了参数的顺序，对于程序的功能也没有任何影响。

关键字参数最厉害的地方在于可以在函数中给参数提供默认值：

```
def hello_3(greeting="Hello", name="world"):
    print "%s, %s!" % (greeting, name)
# 当参数具有默认值的时候，调用的时候就不用提供参数了！可以不提供、提供一些或提供所有的参数:
>>> hello_3()
Hello, world!
>>> hello_3("Greetings")
Greetings, world!
>>> hello_3("Greetings", "universe")
Greetings, universe!
```

可以看到，位置参数这个方法不错，只是在提供名字的时候同时还要提供问候语。但是如果只想提供 `name` 参数，而让 `greeting` 使用默认值该怎么办呢？相信此刻你已经猜到了：

```
>>> hello_3(name="XuHoo")
Hello, XuHoo!
```

很简洁吧？还没完。位置参数和关键字参数是可以联合使用的。把位置参数放置在前面就可以了。如果不这样做，解释器会不知道它们到底是谁(也就是它们应该处的位置)。

注：除非完全清除程序的功能和参数的意义，否则应该避免混合使用位置参数和关键字参数。一般来说，只有在强制要求的参数个数比可修改的具有默认值的参数个数少的时候，才使用上面提到的参数书写方法。

例如， `hello` 函数可能需要名字作为参数，但是也允许用户自定义名字、问候语和标点：

```
def hello_4(name, greeting="Hello", punctuation="!"):
    print "%s, %s%s" % (greeting, name, punctuation)
# 调用函数的方式很多，下面是其中一些:
>>> hello_4("Mars")
Hello, Mars!
>>> hello_4("Mars", "Howdy")
Howdy, Mars!
>>> hello_4("Mars", "Howdy", "...")
Howdy, Mars...
>>> hello_4("Mars", punctuation=".")
Hello, Mars.
>>> hello_4("Mars", greeting="Top of the morning to ya")
Top of the morning to ya, Mars!
>>> hello_4()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  TypeError: hello_4() takes at least 1 argument (0 given)
```

注：如果为 `name` 也赋予默认值，那么最后一个语句就不会产生异常。

很灵活吧？我们也不需要做多少工作。下一节中我们可以做得更灵活。

6.4.4 收集参数

有些时候让用户提供任意数量的参数是很有用的。比如在名字存储程序中(本章前面“为什么我想要修改参数”一节用到的)，用户每次只能存一个名字。如果能像下面这样存储多个名字就更好了：

```
>>> store(data, name1, name2, name3)
# 用户可以给函数提供任意多的参数。实现起来也不难。
# 试着像下面这样定义函数:
def print_params(*params):
    print params
# 这里我只指定了一个参数，但是前面加上了个星号。这是什么意思？让我们用一个参数调用函数看看会发生什么:
>>> print_params("XuHoo")
('XuHoo',)
# 可以看到，结果作为元组打印出来，因为里面有个逗号(长度为1的元组有些奇怪，不是吗)。所以在参数前使用星号就能打印出元组？那么在Params中使用多个参数看看会发生什么:
>>> print_params(1, 2, 3)
(1, 2, 3)
# 参数前的星号将所有值放置在同一个元组中。可以说是将这些值收集起来，然后使用。不知道能不能与普通参数联合使用。让我们再写个函数:
def print_params_2(title, *params):
    print title
    print params
# 试试看
>>> print_params_2("Params:", 1, 2, 3)
Params:
(1, 2, 3)
# 没问题！所以星号的意思就是"收集其余的位置参数"。如果不提供任何供收集的元素，params就是个空元组:
>>> print_params_2("Nothing",)
Nothing
()
# 的确如此，很有用。那么能不能处理关键字参数(也是参数)呢？
>>> print_params_2("XuHoo", something=19)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  TypeError: print_params_2() got an unexpected keyword argument 'something'

# 看来不行。所以我们需要另外一个能处理关键字参数的“收集操作”。那么语法应该怎么写呢？会不会是"**"？
def print_params_3(**params):
    print params
# 至少解释器没有报错。调用一下看看:
>>> print_params_3(x=1, y=2, z=3)
{'y': 2, 'x': 1, 'z': 3}
# 返回的是字典而不是元组。放一起用用看:
def print_params_4(x, y, z=3, *pospar, **keypar):
    print x, y, z
    print pospar
    print keypar
# 和我们期望的结果别无二致:
>>> print_params_4(1, 2, 3, 4, 5, 6, 7, foo=1, bar=2)
1 2 3
(4, 5, 6, 7)
{'foo': 1, 'bar': 2}
>>> print_params_4(1, 2)
1 2 3
()
{}
```

联合使用这些功能，可以做的事情就多了。如果你想知道几种功能联合起来如何工作(或者说是否允许这么做)，那么就自己动手试试看吧(下一节中，会看到 `*` 和 `**` 是怎么用来进行函数调用的，不管是否在函数定义中使用)。

现在回到原来的问题上：怎么实现多个名字同时存储。解决方案如下：

```
def store(data, *full_names):
    for full_name in full_names:
        names = full_name.split()
        if len(names) == 2:
            names.insert(1, "")
        labels = "first", "middle", "last"
        for label, name in zip(labels, names):
            people = lookup(data, label, name)
            if people:
                people.append(full_name) else:
                data[label][name] = [full_name]
# 使用这个函数就像上一节中的只接受一个名字的函数一样简单:
>>> d = {}
>>> init(d)
>>> store(d, "Han Solo")
# 但是现在可以这样使用:
>>> store(d, "Luke Skywalker", "Anakin Skywalker")
>>> lookup(d, "last", "Skywalker")
["Luke Skywalker", "Anakin Skywalker"]
```

6.4.5 参数收集的逆过程

如何将参数收集为元组和字典已经讨论过了，但是事实上，如果使用 `*` 和 `**` 的话，也可以执行相反的操作。那么参数收集的逆过程是什么样？假设有如下函数：

```
def add(x, y):
    return x + y
```

注： *`operator`* 模块中包含此函数的效率更高的版本。

比如说有个包含由两个要相加的数字组成的元组：

```
params = (1, 2)
```

这个过程或多或少有点像我们上一节中介绍的方法的逆过程。不是要收集参数，而是分配它们在“另一端”。使用 `*` 运算符就简单了——不过是在调用而不是在定义时使用：

```
>>> add(*params)
3
```

对于参数列表来说工作正常，只要扩展的部分是最新的就可以。可以使用同样的技术来处理字典——使用双星号运算符。假设之前定义了 `hello_3` ，那么可以这样使用：

```
>>> params = {"name":"Sir Robin", "greeting":"Well met"}
>>> hello_3(**params)
Well met, Sir Robin!
```

在定义或调用函数时使用星号(或者双星号)仅传递元组或字典，所以可能没遇到什么麻烦：

```
>>> def with_stars(**kwds):
...     print kwds["name"], "is", kwds["age"], "year old"
...
>>> def without_stars(kwds):
...     print kwds["name"], "is", kwds["age"], "year old"
...
>>> args = {"name": "XuHoo", "age": 19}
>>> with_stars(**args)
XuHoo is 19 year old
>>> without_stars(args)
XuHoo is 19 year old
```

可以看到，在 `with_stars` 中，我在定义和调用函数时都使用了星号。而在without_stars中两处都没用，但得到了同样的效果。所以星号只在定义函数(允许使用不定数目的参数)或者调用(“分割”字典或者序列)时才有用。

注：使用拼接*(Splicing)*操作符“传递”参数很有用，因为这样一来就不用关心参数的个数之类的问题，例如：

```
def foo(x, y, z, m=0, n=0):
    print x, y, z, m, n
def call_foo(*args, **kwds):
    print "Calling foo!"
foo(*args, **kwds)
```

在调用超类的构造函数时这个方法尤其有用*(*请参见第九章获取更多信息*)*。

6.4.6 练习使用参数

有了这么多种提供和接受参数的方法，很容易犯晕吧！所以让我们把这些方法放在一起举个例子。首先，我定义了一些函数：

```
def story(**kwds):
    return "Once upon a time, there was a " \ "%(job)s called %(name)s. " % kwds
def power(x, y, *others):
    if others:
        print "Received redundant parameters:", others
    return pow(x, y)
def interval(start, stop=None, step=1):
    "Imitates range() for step > 0"
    if stop is None:  # 如果没有为stop指定值……
        start, stop = 0, start  # 指定参数
    result = []
    i = start  # 计算start索引
    while i < stop:  # 直到计算到stop的索引
        result.append(i)  # 将索引添加到result内……
        i += step  # 用stop(>0)增加索引……
    return result # 让我们试一下:
>>> print story(job="king", name="XuHoo")
Once upon a time, there was a king called XuHoo.
>>> print story(name="Sir Robin", job="brave knight")
Once upon a time, there was a brave knight called Sir Robin.
>>> params = {"job": "language", "name": "Python"}
>>> print story(**params)
Once upon a time, there was a language called Python.
>>> del params["job"]
>>> print story(job="stroke of genius", **params)
Once upon a time, there was a stroke of genius called Python.
>>> power(2, 3) 8
>>> power(3, 2) 9
>>> power(y=3, x=2) 8
>>> params = (5,) * 2
>>> power(*params) 3125
>>> power(3, 3, "Hello, world")
Received redundant parameters: ('Hello, world',) 27
>>> interval(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> interval(1, 5)
[1, 2, 3, 4]
>>> interval(3, 12, 4)
[3, 7, 11]
>>> power(*interval(3, 7))
Received redundant parameters: (5, 6) 81
```

这些函数应该多加练习，加以掌握。

6.5 作用域

到底什么是变量？你可以把它们看做是值的名字。在执行 `x=1` 赋值语句后，名称 `x` 引用到值 `1` 上。这就像用字典一样，键引用值，当然，变量和所对应的值用的是个“不可见”的字典。实际上这么说已经很接近真是情况了。内建的 `vars` 函数可以返回这个字典：

```
>>> x = 1
>>> scope = vars()
>>> scope["x"]
1
>>> scope["x"] += 1
>>> x
2
```

注：一般来说， *`vars`* 所返回的字典是不能修改的，因为根据官方*Python*文档的说法，结果是未定义的。换句话说，可能得不到想要的结果。

这类“不可见字典”叫做命名空间或者作用域。那么到底有多少个命名空间？除了全局作用域外，每个函数调用都会创建一个新的作用域：

```
>>> def foo():
        x = 19
...
>>> x = 1
>>> foo()
>>> x
1
```

这里的 `foo` 函数改变(重绑定)了变量 `x` ，但是在最后的时候， `x` 并没有变。这是因为当调用 `foo` 的时候，新的命名空间就被创建了，它作用于 `foo` 内的代码块。赋值语句 `x=19` 只在内部作用域(局部命名空间)起作用，所以它并不影响外部(全局)作用域中的 `x` 。函数内的变量被称为局部变量(local variable，这是与全局变量相反的概念)。参数的工作原理类似于局部变量，所以用全局变量的名字作为参数名并没有问题。

```
>>> def output(x):
        print x
...
>>> x = 1
>>> y = 2
>>> output(y)
2
```

目前为止一切正常。但是如果需要在函数内部访问全局变量怎么办呢？而且只想读取变量的值(也就是说不想重绑定变量)，一般来说是没有问题的：

```
>>> def combine(parameter):
        print parameter + external
...
>>> external = "berry"
>>> combine("Shrub")
Shrubberry
```

注：像这样引用全局变量是很多错误的引发原因。慎重使用全局变量。

屏蔽引发的问题

读取全局变量一般来说并不是问题，但是还是有个会出问题的事情。如果局部变量或者参数的名字和想要访问的去全局变量相同的话，就不能直接访问了。全局变量会被局部变量屏蔽。

如果的确需要的话，可以使用 `globals` 函数获取全局变量值，该函数的近亲是 `vars` ，它可以返回全局变量的字典(`locals` 返回局部变量的字典)。例如，如果前例中有个叫做 `parameter` 的全局变量，那么就不能在 `combine` 函数内部访问该变量，因为你有一个与之同名的参数。必要时，能使用 `globals()["parameter"]` 获取：

```
>>> def combine(parameter):
...     print parameter + globals()["parameter"]
...
>>> parameter = "berry"
>>> combine("Shrub")
Shrubberry
```

接下来讨论重绑定全局变量(使变量引用其他新值)。如果在函数内部将值赋予一个变量，它会自动生成为局部变量——除非告知Python将其声明为全局变量(注意只有在需要的时候才使用全局变量。它们会让代码变得混乱和不灵活。局部变量可以让代码更加抽象，因为它们是在函数中“隐藏”的)。那么怎么才能告诉Python这是一个全局变量呢？

```
>>> x = 1
>>> def change_global():
... global x
...     x = x + 1 ... >>> change_global() >>> x 2
```

小菜一碟！

嵌套作用域

Python的函数是可以嵌套的，也就是说可以将一个函数放在另一个里面(这个话题稍微有点复杂，如果读者刚刚接触函数和作用域，现在可以先跳过)。下面是一个例子：

```
>>> def foo():
... def bar():
... print "Hello, world!" ...     bar()
```

嵌套一般来说并不是那么有用，但它有一个很突出的应用，例如需要一个函数“创建”另一个。也就意味着可以像下面这样(在其他函数内)书写函数：

```
>>> def multiplier(factor):
... def multiplyByFactor(number):
... return number * factor
... return multiplyByFactor
```

一个函数位于另外一个里面，外层函数返回里层函数。也就是说函数本身被返回了，但并没有被调用。重要的是返回的函数还可以访问它的定义所在的作用域。换句话说，它“带着”它的环境(和相关的局部变量)。

每次调用外层函数，它内部的函数都被重新绑定，factor变量每次都有一个新的值。由于Python的嵌套作用域，来自(multiplier的)外部作用域的这个变量，稍后会被内层函数访问。例如：

```
>>> double = multiplier(2) >>> double(5) 10
>>> triple = multiplier(3) >>> triple(3) 9
>>> multiplier(5)(4) 20
```

类似multiplyByFactor函数存储子封闭作用域的行为叫做闭包(closure)。

外部作用域的变量一般来说是不能进行重新绑定的。但在Python3.0中，nonlocal关键字被引入。它和global关键字的使用方法类似，可以让用户对外部作用域(但并非全局作用域)的变量进行赋值。

6.6 递归

前面已经介绍了很多关于创建和调用函数的知识。函数也可以调用其他函数。令人惊讶的是函数可以调用自身，下面将对此进行介绍。

递归这个词对于没接触过程序设计的人来说可能会比较陌生。简单来说就是引用(或调用)自身的意思。来看一个有点幽默的定义：

recur sion \ri-'k&r-zh&n\ n: see recursion.

(递归[名词]：见递归)。

递归的定义(包括递归函数定义)包括它们自身定义内容的引用。由于每个人对递归的掌握程度不同。它可能会让人大伤脑筋，也可能是小菜一碟。为了深入理解它，读者应该买本计算机科学方面的好书，常用Python解释器也能帮助理解。

使用“递归”的幽默定义来定义递归递归一般来说是不可行的，因为那样什么也做不了。我们需要查找递归的意思，结果它告诉我们请参见递归，无穷尽也。一个类似的函数定义如下：

```
def recursion(): return recursion()
```

显然它做不了任何事情——和刚才那个递归的假定义一样没用。运行一下，会发生什么事情？欢迎尝试：不一会，程序直接就崩溃了(发生异常)。理论上讲，它应该永远运行下去。然而每次调用函数都会用掉一点内存，在足够的函数调用发生后(在之前的调用返回后)，空间就不够了，程序会以一个“超过最大递归深度”的错误信息结束。

这类递归叫做无穷递归(infinite recursion)，类似于while True开始的无穷循环，中间没有break或return语句。因为(理论上讲)它永远不会结束。我们想要的是能做一些有用的事情的递归函数。有用的递归函数包含以下几部分：

a.当函数直接返回值时有基本实例(最小可能性问题)；

b.递归实例，包括一个或者多个问题较小部分的递归调用。

这里关键就是讲问题分解为小部分，递归不能永远继续下去，因为它总是以最小可能性问题结束，而这些问题又存储在基本实例中，所以才会让函数调用自身。

但是怎么将其实现呢？做起来没有看起来这么奇怪。就像我刚才说的那样，每次函数被调用时，针对这个调用的新命名空间会被创建，意味着当函数调用“自身”时，实际上运行的是两个不同的函数(或者说是同一个函数具有两个不同的命名空间)。实际上，可以将它想象成和同种类的一个生物进行对话的另一个生物对话。

6.6.1 两个经典：阶乘和幂

本节中，我们会看到两个经典的递归函数。首先，假设想要计算数n的阶乘。n的阶乘定义为n x (n -1) x (n -2) x … x 1。很多数学应用中都会用到它(比如计算将n个人排为一行共有多少种方法)。那么该怎么计算呢？可以使用循环：

```
def factorial(n):
    result = n for i in range(1, n):
        result *= i return result
```

这个方法可行而且容易实现。它的主要过程是：首先，将result赋值到n上，然后result依次与1~n-1的数相乘，最后返回结果。下面来看看使用递归的版本。关键在于阶乘的数学定义，下面就是：

a.1的阶乘是1；

b.大于1的数n的阶乘是n乘n-1的阶乘。

可以看到，这个定义完全符合刚才所介绍的递归的两个条件。

现在考虑如何将定义实现为函数。理解了定义本身以后，实现其实很简单：

```
def factorial(n): if n == 1: return 1
    else: return n * factorial(n-1)
```

这是定义的直接实现。只要记住函数调用factorial(n)是和调用factorial(n-1)不同的实体就行。

考虑另外一个例子。假设需要计算幂，就像内建的pow函数或者**运算符一样。可以用很多种方法定义一个数的(整数)幂。先看一个简单的例子：power(x, n)(x为n的幂次)是x自乘n-1次的结果(所以x用作乘数n次)。所以power(2, 3)是2乘以自身两次：2 x 2 x 2 = 8。

实现很简单：

```
def power(x, n):
    result = 1
    for i in range(n):
        result *= x return result
```

程序很小巧，接下来把它改编为递归版本：

a.对于任意数字来说，power(x, 0)是1；

b.对于任何大于0的数来说，power(x, n)是x乘以(x, n-1)的结果。

同样，可以看到这与简单版本的递归定义的结果相同。

理解定义是最困难的部分——实现起来就简单了：

```
def power(x, n): if n == 0: return 1
    else: return x * power(x, n-1)
```

文字描述的定义再次被转换为了程序语言(Python代码)。

注：如果函数或算法很复杂而且难懂的话，在实现前用自己的话明确地定义一下是很有帮助的。这类使用“准程序语言”编写的程序称为伪代码。

那么递归有什么用呢？就不能用循环代替吗？答案是肯定的，在大多数情况下可以使用循环，而且大多数情况下还会更有效率(至少会高一些)。但是在多数情况下，递归更加易读，有时会大大提高可读性，尤其当读程序的人懂得递归函数的定义的时候。尽管可以避免编写使用递归的程序，但作为程序员来说还是要理解递归算法以及其他人写的递归程序，这也是最基本的。

6.2.2 另外一个经典：二分法查找

作为递归实践的最后一个例子，来看看这个叫做二分法查找(binary search)的算法例子。

你可能玩过一个游戏，通过询问20个问题，被询问者回答是或不是，然后猜测别人在想什么。对于大多数问题来说，都可以将可能性(或多或少)减半。比如已经知道答案是个人，那么可以问“你是不是在想一个女人”，很显然，提问者不会上来就问“你是不是在想约翰·克里斯”——除非提问者会读心术。这个游戏的数学班就是猜数字。例如，被提问者可能在想一个1~100的数字，提问者需要猜中它。当然，提问者可以耐心地猜上100次，但是真正需要才多少次呢？

答案就是只需要问7次即可。第一个问题类似于“数字是否大于50”，如果被提问者回答说数字大于50，那么就问“是否大于75”，然后继续将满足条件的值=等分(排除不满足条件的)，直到找到正确答案。这个不需要太多考虑就能解答出来。

很多其他问题上也能用同样的方法解决。一个很普遍的问题就是查找一个数字是否存在于一个(排过序)的序列中，还要找到具体位置。还可以使用同样的过程。“这个数字是否存在序列正中间的右边”，如果不是的话，“那么是否在第二个1/4范围内(左侧靠右)”，然后这样继续下去。提问者对数字可能存在的位置上下限心里有数，然后每个问题继续切分可能的距离。

这个算法的本身就是递归的定义，亦可用递归实现。让我们首先重看定义，以保证知道自己在做什么：

a.如果上下限相同，那么就是数字所在的位置，返回；

b.否则找到两者的中点(上下限的平均值)，查找数字是在左侧还是在右侧，继续查找数字所在的那半部分。

这个递归例子的关键就是顺序，所以当找到中间元素的时候，只需要比较它和所查找的数字，如果查找数字较大，那么该数字一定在右侧，反之则在左侧。递归部分就是“继续查找数字所在的那半部分”，因为搜索的具体实现可能会和定义中完全相同。(注意搜索的算法返回的是数字应该在的位置——如果它本身不在序列中，那么所返回位置上的其实就是其他数字)

下面来实现一个二分法查找：

```
def search(sequence, number, lower, upper): if lower == upper: assert number == sequen
ce[upper] return upper else:
        middle = (lower + upper) // 2
        if number > sequence[middle]: return search(sequence, number, middle+1, upper)
 else: return search(sequence, number, lower, middle)
```

完全符合定义。如果lower==upper，那么返回upper，也就是上限。注意，程序假设(断言)所查找的数字一定会被找到(number==sequence[upper])。如果没有到达基本实例，先找到middle，检查数字是在左边还是在右边，然后使用新的上下限继续调用递归过程。也可以将限制设为可选以方便用。只要在函数定义的开始部分加入下面的条件语句即可：

```
def search(sequence, number, lower=0, upper=None): if upper is None:   upper = len(seq
uence) - 1 ······
```

如果现在不提供限制，程序会自动设定查找范围为整个序列，看看行不行：

```
>>> seq = [34, 67, 8, 123, 4, 100, 95] >>> seq.sort() >>> seq
[4, 8, 34, 67, 95, 100, 123] >>> search(seq, 34) 2
>>> search(seq, 100) 5
```

但不必这么麻烦，一则可以直接使用列表方法index，如果想要自己实现的话，只要从程序的开始处循环迭代知道找到数字就行了。

当然可以，使用index没问题。但是只使用循环可能效率有点低。刚才说过查找100内的一个数(或位置)，只需要7个问题即可。用循环的话，在最糟糕的情况下要问100个问题。“没什么大不了的”，有人可能会这样想。但是如果列表有100 000 000 000 000 000 000 000 000 000 000 000个元素，要么循环多次(可能对于Python的列表来说这个大小有些不现实)，就“有什么大不了的”了。二分查找法只需要117个问题。很有效吧？(事实上，可观测到的宇宙内的粒子总数是10**87，也就是说只要290个问题就能分辨它们了！)

注：标准库中的*bisect*模块可以非常有效地实现二分查找。

函数式编程

到现在为止，函数的使用方法和其他对象(字符串、数值、序列，等等)基本上一样，它们可以分配给变量、作为参数传递以及从其他函数返回。有些编程语言(比如Scheme或者LISP)中使用函数几乎可以完成所有的事情，尽管在Python(经常会创建自定义的对象——下一章会讲到)中不用那么倚重函数，但也可以进行函数式程序设计。

Python在应对这类“函数式编程”方面有一些有用的函数：map、filter和reduce函数(Python3.0中这些都被移至functools模块中(除此之外还有apply函数。但这个函数被前面讲到的拼接操作符所取代))。map和filter函数在目前版本的Python中并不是特别有用，并且可以使用列表推导式代替。不过读者可以使用map函数将序列中的元素全部传递给一个函数：

```
>>> map(str, range(10))  # Equivalent to [str(i) for i in range(10)]
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
```

filter函数可以基于一个返回布尔值的函数对元素进行过滤。

```
>>> def func(x):
... return x.isalnum()
... >>> seq = ["foo", "x41", "?!", "***"] >>> filter(func, seq)
['foo', 'x41']
```

本例中，使用列表推导式可以不用专门定义一个函数：

```
>>> [x for x in seq if x.isalnum()]
['foo', 'x41']
```

事实上，还有个叫做lambda表达式的特性，可以创建短小的函数("lambda"来源于希腊字母，在数学中表示匿名函数)。

```
>>> filter(lambda x: x.isalnum, seq)
['foo', 'x41']
```

还是列表推导式更易读吧？

reduce函数一般来说不能轻松被列表推导式代替，但是通常用不到这个功能。它会将序列的前两个元素与给定的函数联合使用，并且将它们的返回值和第3个元素继续联合使用，直到整个序列都处理完毕，并且得到一个最终结果。例如，需要计算一个序列的数字的和，可以使用reduce函数加上lambda x,y: x+y(继续使用相同的数字)(事实上，不是使用lambda函数，而是在operator模块引入每个内建运算符的add函数。使用operator模块中的函数通常比用自己的函数更有效率)：

```
>>> numbers = [72, 101, 108, 108, 111, 44, 32, 119, 111, 114, 108, 100, 33] >>> reduce
(lambda x,y: x+y, numbers) 1161
```

当然，这里也可以使用内建函数sum。

6.7 小结

本章介绍了关于抽象的常见知识以及函数的特殊知识。

☑ 抽象：抽象是隐藏多余细节的艺术。定义处理细节的函数可以让程序更抽象。

☑ 函数定义：函数使用def语句定义。它们是由语句组成的块，可以从“外部世界”获取值(参数)，也可以返回一个或多个值作为运算的结果。

☑ 参数：函数从参数中得到需要的信息。也就是函数调用时设定的变量。Python中有两类参数：位置参数和关键字参数。参数在给定默认值时是可选的。

☑ 作用域：变量存储在作用域(也叫做命名空间)中。Python中有两类主要的作用域——全局作用域和局部作用域。作用域可以嵌套。

☑ 递归：函数可以调用自身，如果它这么做了就叫递归。一切用递归实现的功能都可以用循环实现，但是有些时候递归函数更易读。

☑ 函数式编程：Python有一些进行函数性编程的机制。包括lambda表达式以及map、filter和reduce函数。

6.7.1 本章的新函数

本章涉及的新函数如表6-1所示。

表**6-1** 本章的新函数

```
map(func, seq[, seq, ...])                对序列中的每个元素应用函数
filter(func, seq)                         返回其函数为真的元素的列表
reduce(func, seq[, initial])              等同于func(func(func(seq[0], seq[1]), seq[2]
), ...)
sum(seq)                                  返回seq中所有元素的和
apply(func[, args[, kwargs]])             调用函数，可以提供参数
```

6.7.2 接下来学什么

下一章会通过面向对象程序设计，把抽象提升到一个新高度。你将学到如何创建自定义对象的类型(或者说类)，和Python提供的类型(比如字符串、列表和字典)一起使用，以及如何利用这些知识编写出运行更快、更清晰的程序。如果你真正掌握了下一章的内容，编写大型程序会毫不费力。

第七章 更加抽象

> 来源：http://www.cnblogs.com/Marlowes/p/5426233.html
>
> 作者：Marlowes

前几章介绍了Python主要的内建对象类型（数字、字符串、列表、元组和字典），以及内建函数和标准库的用法，还有定义函数的方法。现在看来，还差一点——创建自己的对象。这正是本章要介绍的内容。

为什么要自定义对象呢？建立自己的对象类型可能很酷，但是做什么用呢？使用字典、序列、数字和字符串来创建函数，完成这项工作还不够吗？这样做当然可以，但是创建自己的对象（尤其是类型或者被称为类的对象）是Python的核心概念——非常核心，事实上，Python被成为面向对象的语言(和SmallTalk、C++、Java以及其他语言一样)。本章将会介绍如何创建对象，以及多态、封装、方法、特性、超类以及继承的概念——新知识很多。那么我们开始吧。

注：熟悉面向对象程序设计概念的读者也应该了解构造函数。本章不会提到构造函数，关于它的完整讨论，请参见第九章。

7.1 对象的魔力

在面向对象程序设计中，术语对象(`object`)基本上可以看做数据(特性)以及由一系列可以存取、操作这些数据的方法所组成的集合。使用对像代替全局变量和函数的原因可能有很多。其中对象最重要的有点包括以下几方面。

☑ 多态(Polymorphism)：意味着可以对不同类的对象使用同样的操作，它们会像被“施了魔法一般”工作。

☑ 封装(Encapsulation)：对外部世界隐藏对象的工作细节。

☑ 继承(Inheritance)：以通用的类为基础建立专门的对象。

在许多关于面向对象程序设计的介绍中，这几个概念的顺序是不同的。封装和继承会首先被介绍，因为它们被用作现实世界中的对象的模型。这种方法不错，但是在我看来，面向对象程序设计最有趣的特性是多态。(以我的经历来看)它也是让大多数人犯晕的特性。所以本章会以多态开始，而且这一个概念就足以让你喜欢面向对象程序设计了。

7.1.1 多态

术语多态来自希腊语，意思是“有多种形式”。多态意味着就算不知道变量所引用的对象类型是什么，还是能对它进行操作，而它也会根据对象(或类)类型的不同而表现出不同的行为。例如，假设一个食品销售的商业网站创建了一个在线支付系统。程序会从系统的其他部分(或者以后可能会设计的其他类似的系统)获得一“购物车”中的商品，接下来要做的就是算出总价然后使用信用卡支付。

当你的程序获得商品时，首先想到的可能是如何具体地表示它们。比如需要将它们作为元组接收，像下面这样：

```
("SPAM", 2.50)
```

如果需要描述性标签和价格，这样就够了。但是这个程序还是不够灵活。我们假设网站支持拍卖服务，价格在货物卖出之前会逐渐降低。如果用户能够把对象放入购物车，然后处理结账(你的系统部分)，等价格到了满意的程度后按下“支付”按钮就好了。

但是这样一来简单的元组就不能满足需要了。为了实现这个功能，代码每次询问价格的时候，对象都需要检查当前的价格(通过网络的某些功能)，价格不能固定在元组中。解决起来不难，只要写个函数：

```
# Don't do it
def getPrice(object):
    if isinstance(object, tuple):
        return object[1]
    else:
        return magic_network_method(object)
```

注：这里用 `isinstance` 进行类型/类检查是为了说明一点，类型检查一般来说并不是什么好方法，能不用则不用。函数 `isinstance` 在*7.2.6*节会介绍。

前面的代码中使用 `isinstance` 函数查看对象是否为元组。如果是的话，就返回它的第2个元素，否则会调用一些“有魔力的”网络方法。

假设网络功能部分已经存在，那么问题已经解决了，目前为止是这样。但程序还不是很灵活。如果某些聪明的程序员决定用十六进制数的字符串来表示价格，然后存储在字典中的键"price"下面呢？没问题，只要更新函数：

```
# Don't do it
def getPrice(object): if isinstance(object, tuple): return object[1] elif isinstance(o
bject, dict): return int(objecct["price"]) else: return magic_network_method(object)
```

现在是不是已经考虑到了所有的可能性？但是如果某些人希望为存储在其他键下面的价格增加新的字典呢？那有怎么办呢？可以再次更新 `getPrice` 函数，但是这种工作还要做多长时间？每次有人要实现价格对象的不同功能时，都要再次实现你的模块。但是如果这个模块已经卖出了并且转到了其他更酷的项目中，那要怎么应付客户？显然这是个不灵活且不切实际的实现多种行为的代码编写方式。

那么应该怎么办？可以让对象自己进行操作。听起来很清楚，但是想一下，这样做会轻松很多。每个新的对象类型都可以检索和计算自己的价格并且返回结果，只需向它询问价格即可。这时候多态(在某种程度上还有封装)就要出场了。

1. 多态和方法

程序接收到一个对象，完全不了解该对象的内部实现方式——它可能有多种“形状”。你要做的就是询问价格，这样就够了，实现方法是我们熟悉的：

```
>>> object.getPrice()
2.5
```

绑定到对象特性上面的函数成为方法(method)。我们已经见过字符串、列表和字典方法。实际上多态也已经出现过：

```
>>> "abc".count("a") 1
>>> [1, 2, "a"].count("a") 1
```

对于变量 `x` 来说，不需要知道它是字符串还是列表，就可以调用它的 `count` 方法，不用管它是什么类型(只要你提供了一个字符串作为参数即可)。

让我们做个实验吧。标准库 `random` 中包含 `choice` 函数，可以从序列中随机选出元素。给变量赋值：

```
>>> from random import choice
>>> x = choice(["Hello, world!", [1, 2, "e", "e", 4]])
```

运行后，变量 `x` 可能会包含字符串 `"Hello, world!"` ，也有可能包含列表 `[1, 2, "e", "e", 4]` ——不用关心到底是哪个类型。要关心的就是在变量 `x` 中字符 `e` 出现多少次，而不管 `x` 是字符串还是列表。可以使用刚才的 `count` 函数，结果如下：

```
>>> x.count("e") 1
```

本例中，看来是字符串胜出了(Marlowes:原文上随机选择到的是字符串。 =_=)。但是关键点在于不需要检测类型：只需要知道 `x` 有个叫做 `count` 的方法，带有一个字符作为参数，并且返回整数值就够了。如果其他人创建的对象类也有 `count` 方法，那也无所谓，你只需要像用字符串和列表一样使用该对象就行了。

2. 多态的多种形式

任何不知道对象到底是什么类型，但是又要对对象“做点儿什么”的时候，都会用到多态。这不仅限于方法，很多内建运算符和函数都有多态的性质，考虑下面这个例子：

```
>>> 1 + 2
3
>>> "Fish" + "license"
'Fishlicense'
```

这里的加运算符对于数字(本例中为整数)和字符串(以及其他类型的序列)都能起作用。为说明这一点，假设有个叫做 `add` 的函数，它可以将两个对象相加。那么可以直接将其定义成上面的形式(功能等同但比 `operator` 模块中的 `add` 函数效率低些)。

```
>>> def add(x, y):
...     return x + y
# 对于很多类型的参数都可以用:
>>> add(1, 2)
3
>>> add("Fish", "license")
'Fishlicense'
```

看起来有些傻，但是关键在于参数可以是任何支持加法的对象(注意，这类对象只支持同类的加法。调用 `add(1, "license")` 不会起作用)。如果需要编写打印对象长度消息的函数，只需要对象具有长度(`len` 函数可用)即可。

```
>>> def length_message(x):
...     print "The length of", repr(x), "is", len(x)
```

可以看到，函数中用了 `repr` 函数， `repr` 函数是多态特性的代表之一，可以对任何东西使用。让我们看看：

```
>>> length_message("Fnord")
The length of 'Fnord' is 5
>>> length_message([1, 2, 3])
The length of [1, 2, 3] is 3
```

很多函数和运算符都是多态的——你写的绝大多数程序可能都是，即便你并非有意这样。只要使用多态函数和运算符，就会与“多态”发生关联。事实上，唯一能够毁掉多态的就是使用函数显式地检查类型，比如 `type` 、 `isinstance` 以及 `issubclass` 函数等。如果可能的话，应该尽力避免使用这些毁掉多态的方式。真正重要的是如何让对象按照你所希望的方式工作，不管它是否是正确的类型(或者类)。

注：这里所讨论的多态的形式是*Python*式编程的核心，也是被成为“鸭子类型”*(duck typing)*的东西。这个名词出自俗语“如果它像鸭子一样呱呱大叫……”。有关它的更多信息，请参见 *http://en.wikipedia.org/wiki/Duck_typing*

7.1.2 封装

封装是指向程序中的其他部分隐藏对象的具体实现的原则。听起来有些像多态，也是使用对象而不用知道其内部细节，两者概念有些类似，因为它们都是抽象的原则，它们都会帮助处理程序组件而不用过多关心多余细节，就像函数做的一样。

但是封装并不等同于多态。多态可以让用户对于不知道是什么类(对象类型)的对象进行方法调用，而封装是可以不用关心对象是如何构建的而直接进行调用。听起来还是有些相似？让我们用多态而不用封装写个例子，假设有个叫做`OpenObject`的类(本章后面会学到如何创建类)：

```
>>> o = OpenObject()
# This is how we create objects...
>>> o.setName("Sir Lancelot")
>>> o.getName()
'Sir Lancelot'
```

创建了一个对象(通过像调用函数一样调用类)后，将变量`o`绑定到该对象上。可以使用`setName`和`getName`方法(假设已经由`OpenObject`类提供)。一切看起来都很完美。但是假设变量`o`将它的名字存储在全局变量`globalName`中：

```
>>> globalName
"Sir Lancelot"
```

这就意味着在使用`OpenObject`类的实例时候，不得不关心`globalName`的内容。实际上要确保不会对它进行任何更改：

```
>>> globalName = "Sir XuHoo"
>>> o.getName()
'Sir XuHoo'
```

如果创建了多个`OpenObject`实例的话就会出现问题，因为变量相同，所以可能会混淆：

```
>>> o1 = OpenObject()
>>> o2 = OpenObject()
>>> o1.setName("Robin Hood")
>>> o2.getName()
'Robin Hood'
```

可以看到，设定一个名字后，其他的名字也就自动设定了。这可不是想要的结果。

基本上，需要将对象进行抽象，调用方法的时候不用关心其他的东西，比如它是否干扰了全局变量。所以能将名字“封装”在对象内吗？没问题。可以将其作为特性(attribute)存储。

正如方法一样，特性是作为变量构成对象的一部分，事实上方法更像是绑定到函数上的属性(在本章的7.2.3节中会看到方法和函数重要的不同点)。

如果不用全局变量而用特性重写类，并且重命名为`ClosedObject`，它会像下面这样工作：

```
>>> c = ClosedObject()
>>> c.setName("Sir Lancelot")
>>> c.getName()
'Sir Lancelot'
```

目前为止还不错。但是，值可能还是存储在全局变量中的。那么再创建另一个对象：

```
>>> r = ClosedObject()
>>> r.setName("Sir Robin")
>>> r.getName()
'Sir Robin'
```

可以看到新的对象的名称已经正确设置。这可能正是我们期望的。但是第一个对象怎么样了呢？

```
>>> c.getName()
'Sir Lancelot'
```

名字还在！这是因为对象有它自己的状态(state)。对象的状态由它的特性(比如名称)来描述。对象的方法可以改变它的特性。所以就像是将一大堆函数(方法)捆在一起，并且给予它们访问变量(特性)的权力，它们可以在函数调用之间保持保存的值。

本章后面的“再论私有化”一节也会对Python的封装机制进行更详细的介绍。

7.1.3 继承

继承是另外一个懒惰(褒义)的行为。程序员不想把同一段代码输入好几次。之前使用的函数避免了这种情况，但是现在又有个更微妙的问题。如果已经有了一个类，而又想建立一个非常类似的呢？新的类可能只是添加几个方法。在编写新类时，又不想把旧类的代码全都复制过去。

比如说有个`Shape`类，可以用来在屏幕上画出指定的形状。现在需要创建一个叫做`Rectangle`的类，它不但可以在屏幕上画出指定的形状，而且还能计算该形状的面积。但又不想把Shape里面已经写好的`draw`方法再写一次。那么该怎么办？可以让`Rectangle`从`Shape`类继承方法。在`Rectangle`对象上调用`draw`方法时，程序会自动从`Shape`类调用该方法。(参见7.2.5节)。

7.2 类和类型

现在读者可能对什么是类有了大体感觉——或者已经有些不耐烦听我对它进行更多介绍了。在开始介绍之前，先来认识一下什么是类，以及它和类型又有什么不同(或相同)。

7.2.1 类到底是什么

前面的部分中，类这个词已经多次出现，可以将它或多或少地视为种类或者类型的同义词。从很多方面来说，这就是类——一种对象。所有的对象都属于某一个类，称为类的实例(instance)。

例如，现在请往窗外看，鸟就是“鸟类” 的实例。鸟类是一个非常通用(抽象)的类，具有很多子类：看到的鸟可能属于子类“百灵鸟”。可以将“鸟类”想象成所有鸟的集合，而“百灵鸟类”是其中的一个子集。当一个对象所属的类是另外一个对象所属类的子集时，前者就被成为后者的子类(subclass)，所以“百灵鸟类”是“鸟类”的子类。相反，“鸟类”是“百灵鸟类”的超类(superclass)。

注：日常交谈中，可能经常用复数来描述对象的类，比如 `birds` 或者 `larkes` 。*Python*中，习惯上都使用单数名词，并且首字母大写，比如 `Bird` 和 `Lark` 。

这样一比喻，子类和超类就容易理解了。但是在面向对象程序设计中，子类的关系是隐式的，因为一个类的定义取决于它所支持的方法。类的所有实例都会包含这些方法，所以所有子类的所有实例都有这些方法。定义子类只是个定义更多(也有可能是重载已经存在的)的方法的过程。

例如，鸟类Bird可能支持fly方法，而企鹅类 `Penguin` (`Bird` 的子类)可能会增加个 `eatFish` 方法。当创建 `Penguin` 类时，可能会想要重写(`override`)超类的 `fly` 方法，对于 `Penguin` 的实例来说，这个方法要么什么也不做，要么就产生异常(参见第8章)，因为 `penguin` (企鹅)不会 `fly` (飞)。

注：在旧版本的*Python*中，类和类型之间有很明显的区别。内建的对象是基于类型的，自定义的对象则是基于类的。可以创建类但是不能创建类型。最近版本的*Python*中，事情有了些变化。基本类型和类之间的界限开始模糊了。可以创建内建类型的子类*(*或子类型*)*，而这些类型的行为更类似于类。在越来越熟悉这门语言后会注意到这一点。如果感兴趣的话，第九章中会有关于这方面的更多信息。

7.2.2 创建自己的类

终于来了！可以创建自己的类了！先来看一个简单的类：

```
# 确定使用新式类
__metaclass__ = type

class Person:
    def setName(self, name):
        self.name = Name
    def getName(self):
        return self.name
    def greet(self):
        print "Hello, world! I'm %s" % self.name
```

注：所谓的旧式类和新式类之间是有区别的。除非是*Python3.0*之前版本中默认附带的代码，否则再继续使用旧式类已无必要。新式类的语法中，需要在模块或者脚本开始的地方放置赋值语句 `__metaclass__ = type` *(*并不会在每个例子中显式地包含这行语句*)*。除此之外也有其他

的方法，例如继承新式类(比如`object`)。后面马上就会介绍继承的知识。在*Python3.0*中，旧式类的问题不用再担心，因为它们根本就不存在了。请参见第九章获取更多信息。

这个例子包含3个方法定义，除了它们是写在`class`语句里面外，一切都像是函数定义。`Person`当然是类的名字。`class`语句会在函数定义的地方创建自己的命名空间(参见7.2.4节)。一切看起来都挺好，但是那个`self`参数看起来有点奇怪。它是对于对象自身的引用。那么它是什么对象？让我们创建一些实例看看：

```
>>> foo = Person()
>>> bar = Person()
>>> foo.setName("Luke Skywalker")
>>> bar.setName("Anakin Skywalker")
>>> foo.greet()
Hello, world! I'm Luke Skywalker
>>> bar.greet()
Hello, world! I'm Anakin Skywalker
```

好了，例子一目了然，应该能说明`self`的用处了。在调用`foo`的`setName`和`greet`函数时，`foo`自动将自己作为第一个参数传入函数中——因此形象的命名为`self`。对于这个变量，每个人可能都会有自己的叫法，但是因为它总是对象自身，所以习惯上总是叫做`self`。

显然这就是`self`的用处和存在的必要性。没有它的话，成员方法就没法访问他们要对其特性进行操作的对象本身了。

和之前一样，特性是可以在外部访问的：

```
>>> foo.name 'Luke Skywalker'
>>> bar.name = "Yoda"
>>> bar.greet()
Hello, world! I'm Yoda
```

注：如果知道*`foo`*是*`Person`*的实例的话，那么还可以把*`foo.greet()`*看作*`Person.greet(foo)`*方便的简写。

7.2.3 特性、函数和方法

(在前面提到的)`self`参数事实上正是方法和函数的区别。方法(更专业一点可以成为绑定方法)将它们的第一个参数绑定到所属的实例上，因此您无需显式提供该参数。当然也可以将特性绑定到一个普通函数上，这样就不会有特殊的`self`参数了：

```
>>> class Class:
...     def method(self):
...         print "I have a self!"

>>> def function():
...     print "I don't..."

>>> instance = Class()
>>> instance.method()
I have a self!
>>> instance.method = function
>>> instance.method()
I don't...
```

注意，`self`参数并不依赖于调用方法的方式，前面我们使用的是`instance.method`(实例.方法)的形式，可以随意使用其他变量引用同一个方法：

```
>>> class Bird:
...     song = "Squaawk!"
...     def sing(self):
...         print self.song
>>> bird = Bird()
>>> bird.sing()
Squaawk!
>>> brid.song
>>> birdsong = bird.sing()
Squaawk!
```

尽管最后一个方法调用起来与函数调用十分相似，但是变量`birdsong`引用绑定方法(第九章中，将会介绍类是如何调用超类方法的(具体来说就是超类的构造器)。这些方法直接通过类调用，他们没有绑定自己的`self`参数到任何东西上，所以叫做非绑定方法)`bird.sing`上，也就意味着这还是会对`self`参数进行访问(也就是说，它仍旧绑定到类的相同实例上)。

再论私有化

默认情况下，程序可以从外部访问一个对象的特性。再次使用前面讨论过的相关封装的例子：

```
>>> c.name 'Sir Lancelot'
>>> c.name = 'Sir Gumby'
>>> c.getName() 'Sir Gumby'
```

有些程序员觉得这样做是可以的，但是有些人(比如SmallTalk之父，SmallTalk的对象特性只允许由同一个对象的方法访问)觉得这样做就破坏了封装的原则。他们认为对象的状态对于外部应该是完全隐藏(不可访问)的。有人可能会奇怪为什么他们会站在如此极端的立场上。每个对象管理自己的特性还不够吗？为什么还要对外部世界隐藏呢？毕竟如果能直接使用`ClosedObject`的`name`特性的话就不用使用`setName`和`getName`方法了。

关键在于其他程序员可能不知道(可能也不应该知道)你的对象内部的具体操作。例如，`ClosedObject`可能会在其他对象更改自己的名字的时候，给一些管理员发送邮件消息。这应该是`setName`方法的一部分。但是如果直接使用`c.name`设定名字会发生什么？什么都没

发生，Email也没发出去。为了避免这类事情的发生，应该使用私有(private)特性，这是外部对象无法访问，但 `getName` 和 `setName` 等访问器(accessor)能够访问的特性。

注：第九章中，将会介绍有关属性*(property)*的只是，它是访问器最有力的替代者。

Python并不直接支持私有方式，而是要靠程序员自己把握在外部进行特性修改的时机。毕竟在使用对象前应该知道如何使用。但是，可以用一些小技巧达到私有特性的效果。

为了让方法或者特性变为私有(从外部无法访问)，只要在它的名字前面加上双下划线即可：

```
class Secretive():
    def __inaccessible(self):
        print "Bet you can't see me..."

    def accessible(self):
        print "The secret message is:"
        self.__inaccessible()
```

现在 `__inaccessible` 从外界是无法访问的，而在类内部还能使用(比如从 `accessible`)访问：

```
>>> s = Secretive()
>>> s.__inaccessible()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  AttributeError: Secretive instance has no attribute '__inaccessible'
>>> s.accessible()
The secret message is:
Bet you can't see me...
```

尽管双下划线有些奇怪，但是看起来像是其他语言中的标准的私有方法。真正发生的事情才是不标准的。类的内部定义中，所有以双下划线开始的名字都被“翻译”成前面加上单下划线和类名的形式：

```
>>> Secretive._Secretive__inaccessible <unbound method Secretive.__inaccessible>
```

在了解这些幕后的事情后，实际上还能在类外访问这些私有方法，尽管不应该这么做：

```
>>> s._Secretive__inaccessible()
Bet you can't see me...
```

简而言之，确保其他人不会访问对象的方法和特性是不可能的，但是这类”名称变化术“是他们不应该访问这些函数或者特性的强有力信号。

如果不需要使用这种方法但是又想让其他对象不要访问内部数据，那么可以使用单下划线。这不过是个习惯，但的确有实际效果。例如，前面有下划线的名字都不会被带星号的import语句(`from module import *`)导入(有些语言支持多层次的成员变量(特性)私有性。比如Java就支持4种级别。尽管单下划线在某种程度上给出两个级别的私有性，但Python并没有真正的私有化支持)。

7.2.4 类的命名空间

下面的两个语句(几乎)等价：

```
def foo(x):
    return x * x
foo = lambda x: x * x
```

两者都创建了返回参数平方的函数，而且都将变量`foo`绑定到函数上。变量`foo`可以在全局(模块)范围进行定义，也可处于局部的函数或方法内。定义类时，同样的事情也会发生，所有位于`class`语句中的代码都在特殊的命名空间中执行——类命名空间(`class namespace`)。这个命名空间可由类内所有成员访问。并不是所有Python程序员都知道类的定义其实就是执行代码块，这一点非常有用，比如，在类的定义区并不只限定只能使用self语句：

```
>>> class C:
...     print "Class C being defined..."
...
Class C being defined...
>>>
```

看起来有点傻，但是看看下面的：

```
>>> class MemberCounter:
...     members = 0
...     def init(self):
...         MemberCounter.members += 1
...
>>> m1 = MemberCounter()
>>> m1.init()
>>> MemberCounter.members
1
>>> m2 = MemberCounter()
>>> m2.init()
>>> MemberCounter.members
2
```

上面的代码中，在类作用域内定义了一个可供所有成员(实例)访问的变量，用来计算类的成员数量。注意`init`用来初始化所有实例：第九章中，我会让这一过程自动化(即把它变成一个适当的构造函数)。

就像方法一样，类作用域内的变量也可以被所有实例访问：

```
>>> m1.members 2
>>> m2.members 2
```

那么在实例中重绑定`members`特性呢？

```
>>> m1.members = "Two"
>>> m1.members 'Two'
>>> m2.members
2
```

新 `members` 值被写到了 `m1` 的特性中，屏蔽了类范围内的变量。这跟函数内的局部和全局变量的行为十分类似，就像第六章讨论的”屏蔽的问题“。

7.2.5 指定超类

就像本章前面我们讨论的一样，子类可以扩展超类的定义。将其他类名写在 `class` 语句后的圆括号内可以指定超类：

```
class Filter():
    def init(self):
        self.blocked = []
        def filter(self, sequence):
            return [x for x in sequence if x not in self.blocked]
class SPAMFilter(Filter):
    # SPAMFilter是Filter的子类

    def init(self):
        # 重写Filter超类中的init方法
        self.blocked = ["SPAM"]
```

`Filter` 是个用于过滤序列的通用类，事实上它不能过滤任何东西：

```
>>> f = Filter()
>>> f.init()
>>> f.filter([1, 2 ,3])
[1, 2, 3]
```

`Filter` 类的用处在于它可以用作其他类的基类(超类)，比如 `SPAMFilter` 类，可以将序列中的 `'SPAM'` 过滤出去。

```
>>> s = SPAMFilter()
>>> s.init()
>>> s.filter(["SPAM", "SPAM", "SPAM", "SPAM", "eggs", "bacon", "SPAM"])
['eggs', 'bacon']
```

注意 `SPAMFilter` 定义的两个要点。

☑ 这里用提供新定义的方式重写了 `Filter` 的 `init` 定义。

☑ `filter` 方法的定义是从 `Filter` 类中拿过来(继承)的，所以不用重写它的定义。

第二个要点揭示了继承的用处：我可以写一大堆不同的过滤类，全部都从 `Filter` 继承，每一个我都可以使用已经实现的 `filter` 方法。这就是前面提到过的有用的懒惰。

7.2.6 检查继承

如果想要查看一个类是否是另一个的子类，可以使用内建的 `issubclass` 函数：

```
>>> issubclass(SPAMFilter, Filter)
True
>>> issubclass(Filter, SPAMFilter)
False
```

如果想要知道已知的基类(们)，可以直接使用它的特殊特性 `__bases__` 。

```
>>> SPAMFilter.__bases__ (<class __main__.Filter at 0x7fa160e4a4c8>,)
>>> Filter.__bases__ ()
```

同样，还能用使用 `isinstance` 方法检查一个对象是否是一个类的实例：

```
>>> s = SPAMFilter()
>>> isinstance(s, SPAMFilter)
True
>>> isinstance(s, Filter)
True
>>> isinstance(s, str)
False
```

注：使用 *`isinstance`* 并不是个好习惯，使用多态会更好一些。

可以看到，s是 `SPAMFilter` 类的(直接)实例，但是它也是 `Filter` 类的间接实例，因为 `SPAMFilter` 是 `Filter` 的子类。另外一种说法就是 `SPAMFilter` 类就是 `Filters` 类。可以从前一个例子中看到， `isinstance` 对于类型也起作用，比如字符串类型(`str`)。

如果只想知道一个对象属于哪个类，可以使用 `__class__` 特性：

```
>>> s.__class__
<class __main__.SPAMFilter at 0x7fa160e4a530>
```

注：如果使用 *`__metaclass__ = type`* 或从 *`object`* 继承的方式来定义新式类，那么可以使用 *`type(s)`* 查看实例所属的类。

7.2.7 多个超类

可能有的读者注意到了上一节中的代码有些奇怪：也就是 `__bases__` 这个复数形式。而且文中也提到过可以找到一个新的基类(们)，也就按暗示它的基类可能会多余一个。事实上就是这样，建立几个新的类来试试看：

```
class Calculator:
    def calculate(self, expression):
        self.value = eval(expression)
class Talker:
    def talk(self):
        print "Hi, my value is", self.value
class TalkingCalculator(Calculator, Talker):
    pass
```

子类(`TalkingCalculator`)自己不做任何事，它从自己的超类继承所有的行为。它从`Calculator`类那里继承`calculate`方法，从`Talker`类那里继承`talk`方法，这样它就成了会说话的计算器(talking calculator)。

```
>>> tc = TalkingCalculator()
>>> tc.calculate("1 + 2 * 3")
>>> tc.talk()
Hi, my value is 7
```

这种行为称为多重继承(multiple inheritance)，是个非常有用的工具。但除非读者特别熟悉多重继承，否则应该尽量避免使用，因为有些时候会出现不可预见的麻烦。

当使用多重继承时，有个需要注意的地方。如果一个方法从多个超类继承(也就是说你有两个具有相同名字的不同方法)，那么必须要注意一下超类的顺序(在`class`语句中)：先继承的类中的方法会重写后继承的类中的方法。所以如果前例中`Calculator`类也有个叫做`talk`的方法，那么它就会重写`Talker`的`talk`方法(使其不可访问)。如果把它们的顺序调过来，像下面这样：

```
class TalkingCalculator(Talker, Calculator):
    pass
```

就会让`Talker`的`talk`方法可用了。如果超类们共享一个超类，那么在查找给定方法或者属性时访问超的顺序称为MRO(Method Resolution Order， 方法判定顺序)，使用的算法相当复杂。幸好，它工作得很好，所以不用过多关心。

7.2.8 接口与内省

“接口”的概念与多态有关。在处理多态对象时，只要关心它的接口(或称“协议”)即可，也就是公开的方法和特性。在Python中，不用显式地指定对象必须包含哪些方法才能作为参数接收。例如，不用(像在Java中一样)显式地编写接口，可以在使用对象的时候假定它可以实现你所要求的行为。如果它不能实现的话，程序就会失败。

一般来说只需要让对象符合当前的接口(换句话说就是实现当前方法)，但是还可以更灵活一些。除了调用方法然后期待一切顺利之外，还可检查所需方法是否已经存在。如果不存在，就需要做些其他事情：

```
>>> hasattr(tc, "talk")
True
>>> hasattr(tc, "fnord")
False
```

注：*`callable`*函数在*Python3.0*中已不再可用。可以使用*`hasattr(x, "__call__")`*来代替*`callable(x)`*。

这段代码使用了 `getattr` 函数，而没有在 `if` 语句内使用 `hasattr` 函数直接访问特性，`getattr` 函数允许提供默认值(本例中为 `None`)，以便在特性不存在时使用，然后对返回的对象使用 `callable` 函数。

注：与 *`getattr`* 相对应的函数是 *`setattr`*，可以用来设置对象的特性：

```
>>> setattr(tc, "name", "Mr. XuHoo")
>>> tc.name 'Mr. XuHoo'
```

如果要查看对象内所有存储的值，那么可以使用 `__dict__` 特性。如果真的想要找到对象是由什么组成的，可以看看 `inspect` 模块。这是为那些想要编写对象浏览器(以图形方式浏览Python对象的程序)以及其他需要类似功能的程序的高级用户准备的。关于对象和模块的更多信息，可以参见10.2节。

7.3 一些关于面向对象设计的思考

关于面向对象设计的书籍已经有很多，尽管这并不是本书所关注的主题，但是还是给出一些要点。

☑ 将属于一类的对象放在一起。如果一个函数操纵一个全局变量，那么两者最好都在类内作为特性和方法出现。

☑ 不要让对象过于亲密。方法应该只关心自己实例的特性。让其他实例管理自己的状态。

☑ 要小心继承，尤其是多重继承。继承机制有时很有用，但也会在某些情况下让事情变得过于复杂。多继承难以正确使用，更加难以调试。

☑ 简单就好。让你的方法小巧。一般来说，多数方法都应能在30秒内被读完(以及理解)，尽量将代码行数控制在一页或者一屏之内。

当考虑需要什么类以及类要有什么方法时，应该尝试下面的方法。

(1)写下问题的描述(程序要做什么)，把所有的名词、动词和形容词加下划线。

(2)对于所有名词，用作可能的类。

(3)对于所有动词，用作可能的方法。

(4)对于所有形容词，用作可能的特性。

(5)把所有方法和特性分配到类。

现在已经有了面向对象模型的草图了。还可以考虑类和对象之间的关系(比如继承或协作)以及它们的作用，可以用以下步骤精炼模型。

(1)写下(或者想象)一系列的使用实例，也就是程序应用时的场景，试着包括所有的功能。

(2)一步步考虑每个使用实例，保证模型包括所有需要的东西。如果有些遗漏的话就添加进来。如果某处不太正确则改正。继续，直到满意为止，

当认为已经有了可以应用的模型时，那就可以开工了。可能需要修正自己的模型，或者是程序的一部分。幸好，在Python中不用过多关心这方面的事情，因为很简单，只要投入进去就行(如果需要面向对象程序设计方面的更多指导，请参见第十九章推荐的书目)。

7.4 小结

本章不仅介绍了更多关于Python语言的信息，并且介绍了几个可能完全陌生的概念。下面总结一下。

☑ 对象：对象包括特性和方法。特性只是作为对象的一部分变量，方法则是存储在对象内的函数。(绑定)方法和其他函数的区别在于方法总是将对象作为自己的第一个参数，这个参数一般称为self。

☑ 类：类代表对象的集合(或一类对象)，每个对象(实例)都有一个类。类的主要任务是定义它的实例会用到的方法。

☑ 多态：多态是实现将不同类型和类的对象进行同样对待的特性——不需要知道对象属于哪个类就能调用方法。

☑ 封装：对象可以将它们的内部状态隐藏(或封装)起来。在一些语言中，这意味着对象的状态(特性)只对自己的方法可用。在Python中，所有的特性都是公开可用的，但是程序员应该在直接访问对象状态时谨慎行事，因为他们可能无意中使得这些特性在某些方面不一致。

☑ 继承：一个类可以是一个或者多个类的子类。子类从超类继承所有方法。可以使用多个超类，这个特性可以用来组成功能的正交部分(没有任何联系)。普通的实现方式是使用核心的超类和一个或者多个混合的超类。

☑ 接口和内省：一般来说，对于对象不用探讨过深。程序员可以靠多态调用自己需要的方法。不过如果想要知道对象到底有什么方法和特性，有些函数可以帮助完成这项工作。

☑ 面向对象设计：关于如何(或者说是否应该进行)面向对象设计有很多的观点。不管你持什么观点，完全理解这个问题，并且创建容易理解的设计是很重要的。

7.4.1 本章的新函数

本章涉及的新函数如表7-1所示。

表**7-1** 本章的新函数

```
callable(object)                   确定对象是否可调用(比如函数或者方法)
getattr(object, name[ ,default])   确定特性的值，可选择提供默认值
hasattr(object, name)              确定对象是否有给定的特性
isinstance(object, class)          确定对象是否是类的实例
issubclass(A, B)                   确定A是否为B的子类
random.choice(sequence)            从非空序列中随机选择元素
setattr(object, name, value)       设定对象的给定特性为value
type(object)                       返回对象的类型
```

7.4.2 接下来学什么

前面已经介绍了许多关于创建自己的对象以及自定义对象的作用。在轻率地进军Python特殊方法的魔法阵(第九章)之前，让我们先喘口气，看看介绍异常处理的简短的一章。

第八章 异常

> 来源：http://www.cnblogs.com/Marlowes/p/5428641.html
>
> 作者：Marlowes

在编写程序的时候，程序员通常需要辨别事件的正常过程和异常(非正常)的情况。这类异常事件可能是错误(比如试图除以 `0`)，或者是不希望经常发生的事情。为了能够处理这些异常事件，可以在所有可能发生这类事件的地方都使用条件语句(比如让程序检查除法的分母是否为零)。但是，这么做可能不仅会没效率和不灵活，而且还会让程序难以阅读。你可能会想直接忽略这些异常事件，期望它们永不发生，但Python的异常对象提供了非常强大的替代解决方案。

本章介绍如何创建和引发自定义的异常，以及处理异常的各种方法。

8.1 什么是异常

Python用异常对象(exception object)来表示异常情况。遇到错误后，会引发异常。如果异常对象并未被处理或捕捉，程序就会用所谓的回溯(traceback， 一种错误信息)终止执行：

```
>>> 1 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  ZeroDivisionError: integer division or modulo by zero
```

如果这些错误信息就是异常的全部功能，那么它也就不必存在了。事实上，每个异常都是一些类(本例中是 `ZeroDivisionError`)的实例，这些实例可以被引发，并且可以用很多种方法进行捕捉，使得程序可以捉住错误并且对其进行处理，而不是让整个程序失效。

8.2 按自己的方式出错

异常可以在某些东西出错的时候自动引发。在学习如何处理异常之前，先看一下自己如何引发异常，以及创建自己的异常类型。

8.2.1 `raise` 语句

为了引发异常，可以使用一个类(应该是 `Exception` 的子类)或者实例参数调用 `raise` 语句。使用类时，程序会自动创建类的一个实例。下面是一些简单的例子，使用了内建的 `Exception` 的异常类：

```
>>> raise Exception
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  Exception
>>> raise Exception("hyperdrive overload")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module> \
  Exception: hyperdrive overload
```

第一个例子raise Exception引发了一个没有任何有关错误信息的普通异常。后一个例子中，则添加了错误信息hyperdrive overload。

内建的异常类有很多。Python库参考手册的Built-in Exceptions一节中有关与它们的描述。用交互式解释器也可以分析它们，这些内建异常都可以在`exceptions`模块(和内建的命名空间)中找到。可以使用`dir`函数列出模块内容，这部分会在第十章中讲到：

```
>>> import exceptions
>>> dir(exceptions)
['ArithmeticError', 'AssertionError', 'AttributeError', ...]
```

读者的解释器中，这个名单可能要长得多——出于对易读性的考虑，这里删除了大部分名字，所有这些异常都可以用在`raise`语句中：

```
>>> raise ArithmeticError
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  ArithmeticError
```

表8-1描述了一些最重要的内建异常类：

表8-1 一些内建异常类

```
Exception                    所有异常的基类
AttributeError                特性引用或赋值失败时引发
IOError                      试图打开不存在文件(包括其他情况)时引发
IndexError                        在使用序列中不存在的索引时引发
KeyError                     在使用映射中不存在的键时引发
NameError                     在找不到名字(变量)时引发
SyntaxError                   在代码为错误形式时引发
TypeError                    在内建操作或者函数应用于错误类型的对象时引发
ValueError                    在内建操作或者函数应用于正确类型的对象，但是该对象使用不合适的值时引
发
ZeroDivisionError               在除法或者模除操作的第二个参数为0时引发
```

8.2.2 自定义异常类

尽管内建的异常类已经包括了大部分的情况，而且对于很多要求都已经足够了，但是有些时候还是需要创建自己的异常类。比如在超光速推进装置过载(hyperdrive overload)的例子中，如果能有个具体的`HyperDriveError`类来表示超光速推进装置的错误状况是不是更自然一些？

错误信息是足够了，但是会在8.3节中看到，可以根据异常所在的类，选择性地处理当前类型的异常。所以如果想要使用特殊的错误处理代码处理超光速推进装置的错误，那么就需要一个独立于`exceptions`模块的异常类。

那么如何创建自己的异常类呢？就像其他类一样，只是要确保从`Exception`类继承(不管是间接还是直接，也就是说继承其他的内建异常类也是可以的)。那么编写一个自定义异常类基本上就像下面这样：

```
class SomeCustomException(Exception):
    pass
```

还不能做太多事，对吧？(如果你愿意，也可以向你的异常类中增加方法)

8.3 捕捉异常

前面曾经提到过，关于异常的最有意思的地方就是可以处理它们(通常叫做诱捕或者捕捉异常)。这个功能可以使用`try/except`语句来实现。假设创建了一个让用户输入两个数，然后进行相除的程序，像下面这样：

```
x = input("Enter the first number: ")
y = input("Enter the second number: ")
print x / y
```

程序工作正常，假如用户输入0作为第二个数

```
Enter the first number: 10 Enter the second number: 0
Traceback (most recent call last):
  File "/home/marlowes/MyPython/My_Exception.py", line 6, in <module>
    print x / y
ZeroDivisionError: integer division or modulo by zero
```

为了捕捉异常并且做出一些错误处理(本例中只是输出一些更友好的错误信息)，可以这样重写程序：

```
try:
    x = input("Enter the first number: ")
    y = input("Enter the second number: ")
    print x / y
except ZeroDivisionError:
    print "The second number can't be zero!"
```

看起来用`if`语句检查`y`值会更简单一些，本例中这样做的确很好。但是如果需要给程序加入更多除法，那么就得给每个除法加个if语句。而且使用`try/except`的话只需要一个错误处理器。

注：如果没有捕捉异常，它就会被“传播”到调用的函数中。如果在那里依然没有捕获，这些异常就会“浮”到程序的最顶层，也就是说你可以捕捉到在其他人的函数中所引发的异常。有关这方面的更多信息，请参见*8.10*节。

看，没参数

如果捕捉到了异常，但是又想重新引发它(也就是说要传递异常，不进行处理)，那么可以调用不带参数的`raise`(还能在捕捉到异常时显式地提供具体异常，在8.6节会对此进行解释)。

举个例子吧，看看这么做多么有用：考虑一下一个能“屏蔽”`ZeroDivisionError`(除零错误)的计算器类。如果这个行为被激活，那么计算器就打印错误信息，而不是让异常传播。如果在与用户交互的过程中使用，那么这就有用了，但是如果是在程序内部使用，引发异常会更好些。因此“屏蔽”机制就可以关掉了，下面是这样一个类的代码：

```
class MuffledCalculator():

    muffled = False
    def calc(self, expr):
        try:
            return eval(expr)
        except ZeroDivisionError:
            if self.muffled:
                print "Division by zero is illegal"
            else:
                raise
```

注：如果除零行为发生而屏蔽机制被打开，那么`calc`方法会*(*隐式地*)*返回`None`。换句话说，如果打开了屏蔽机制，那么就不应该依赖返回值。

下面是这个类的用法示例，分别打开和关闭了屏蔽：

```
>>> calculator = MuffledCalculator()
>>> calculator.calc("10 / 2")
5
>>> calculator.calc("10 / 0")
Traceback (most recent call last):
  File "/home/marlowes/MyPython/My_Exception.py", line 28, in <module> calculator.calc
("10 / 0")
  File "/home/marlowes/MyPython/My_Exception.py", line 19, in calc return eval(expr)
  File "<string>", line 1, in <module> ZeroDivisionError: integer division or modulo b
y zero >>> calculator.muffled = True >>> calculator.calc("10 / 0")
Division by zero is illegal
```

当计算器没有打开屏蔽机制时，`ZeroDivisionError`被捕捉但已传递了。

8.4 不止一个 `except` 子句

如果运行上一节的程序并且在提示符后面输入非数字类型的值，就会产生另一个异常：

```
Enter the first number: 10
Enter the second number: "Hello, world!"
Traceback (most recent call last):
  File "/home/marlowes/MyPython/My_Exception.py", line 8, in <module>
    print x / y
TypeError: unsupported operand type(s) for /: 'int' and 'str'
```

因为`except`子句只寻找`ZeroDivisionError`异常，这次的错误就溜过了检查并导致程序终止。为了捕捉这个异常，可以直接在同一个`try/except`语句后面加上另一个`except`子句：

```
try:
    x = input("Enter the first number: ")
    y = input("Enter the second number: ")
    print x / y
except ZeroDivisionError:
    print "The second number can't be zero!"
except TypeError:
    print "That wasn't a number, was it?"
```

这次用`if`语句实现可就复杂了。怎么检查一个值是否能被用在除法中？方法很多，但是目前最好的方式是直接将值用来除一下看看是否奏效。

还应该注意到，异常处理并不会搞乱原来的代码，而增加一大堆`if`语句检查可能的错误情况会让代码相当难读。

8.5 用一个块捕捉两个异常

如果需要用一个块捕捉多个类型异常，那么可以将它们作为元组列出，像下面这样：

```
try:
    x = input("Enter the first number: ")
    y = input("Enter the second number: ")
    print x / y
except (ZeroDivisionError, TypeError, NameError):
    print "Your numbers were bogus..."
```

上面的代码中，如果用户输入字符串或者其他类型的值，而不是数字，或者第2个数为0，都会打印同样的错误信息。当然，只打印一个错误信息并没有什么帮助。另外一个方法就是继续要求输入数字直到可以进行除法运算为止。8.8节中会介绍如何实现这一功能。

注意，`except`子句中异常对象外面的圆括号很重要。忽略它们是一种常见的错误，那样你会得不到想要的结果。关于这方面的解释，请参见8.6节。

8.6 捕捉对象

如果希望在`except`子句中访问异常对象本身，可以使用两个参数(注意，就算要捕捉到多个异常，也只需向`except`子句提供一个参数——一个元组)。比如，如果想让程序继续运行，但是又因为某种原因想记录下错误(比如只是打印给用户看)，这个功能就很有用。下面的示例程序会打印异常(如果发生的话)，但是程序会继续运行：

```
try:
    x = input("Enter the first number: ")
    y = input("Enter the second number: ")
    print x / y
except (ZeroDivisionError, TypeError), e:
    print e
```

(在这个小程序中，`except`子句再次捕捉了两种异常，但是因为你可以显式地捕捉对象本身，所以异常可以打印出来，用户就能看到发生什么(8.8节会介绍一个更有用的方法)。——译者注)

注：在*Python3.0*中，*`except`*子句会被写作*`except (ZeroDivisionError, TypeError) as e`*。

8.7 眞正的捕捉

就算程序能处理好几种类型的异常，但是有些异常还会从眼皮地下溜走。比如还用那个除法程序来举例，在提示符下面直接按回车，不输入任何东西，会的到一个类似下面这样的错误信息(栈跟踪)：

```
Traceback (most recent call last):
  File "/home/marlowes/MyPython/My_Exception.py", line 33, in <module> x = input("Ente
r the first number: ")
  File "<string>", line 0
  ^ SyntaxError: unexpected EOF while parsing
```

这个异常逃过了`try/except`语句的检查，这很正常。程序员无法预测会发生什么，也不能对其进行准备。在这些情况下，与其用那些并非捕捉这些异常的`try/except`语句隐藏异常，还不如让程序立刻崩溃。

但是如果眞的想用一段代码捕捉所有异常，那么可以在except子句中忽略所有的异常类：

```
try:
    x = input("Enter the first number: ")
    y = input("Enter the second number: ")
    print x / y
except:
    print "Something wrong happened..."
```

现在可以做任何事情了：

```
Enter the first number: "This" is *completely* illegal 123 Something wrong happened...
```

警告：像这样捕捉所有异常是危险的，因为它会隐藏所有程序员未想到并且未做好准备处理的错误。它同样会捕捉用户终止执行的*Ctrl+C*企图，以及用 `sys.exit` 函数终止程序的企图，等等。这时使用 `except Exception, e` 会更好些，或者对异常对象 `e` 进行一些检查。

8.8 万事大吉

有些情况中，没有坏事发生时执行一段代码是很有用的；可以像对条件和循环语句那样，给 `try/except` 语句加个 `else` 子句：

```
try:
    print "A simple task"
except:
    print "What? Something went wrong?"
else:
    print "Ah... It went as planned."
```

运行之后会的到如下输出：

```
A simple task
Ah... It went as planned.
```

使用 `else` 子句可以实现在8.5节中提到的循环：

```
while True:
    try:
        x = input("Enter the first number: ")
        y = input("Enter the second number: ")
        value = x / y
        print "x / y is", value
    except:
        print "Invalid input. Please try again."
    else:
        break
```

这里的循环只有在没有异常引发的情况下才会退出(由 `else` 子句中的 `break` 语句退出)。换句话说，只要有错误发生，程序会不断要求重新输入。下面是一个例子的运行情况：

```
Enter the first number: 1
Enter the second number: 0
Invalid input. Please try again.
Enter the first number: "foo"
Enter the second number: "bar"
Invalid input. Please try again.
Enter the first number: baz
Invalid input. Please try again.
Enter the first number: 10
Enter the second number: 2
x / y is 5
```

之前提到过，可以使用空的 `except` 子句来捕捉所有 `Exception` 类的异常(也会捕捉其所有子类的异常)。百分之百捕捉到所有的异常是不可能的，因为 `try/except` 语句中的代码可能会出现问题，比如使用旧风格的字符串异常或者自定义的异常类不是 `Exception` 类的子类。不过如果需要使用 `except Exception` 的话，可以使用8.6节中的技巧在除法程序中打印更加有用的错误信息：

```
while True:
    try:
        x = input("Enter the first number: ")
        y = input("Enter the second number: ")
        value = x / y
        print "x / y is", value
    except Exception, e:
        print "Invalid input:", e
        print "Please try again"
    else:
        break
```

下面是示例运行：

```
Enter the first number: 1 Enter the second number: 0
Invalid input: integer division or modulo by zero
Please try again
Enter the first number: "x"
Enter the second number: "y"
Invalid input: unsupported operand type(s) for /: 'str' and 'str' Please try again
Enter the first number: quuux
Invalid input: name 'quuux' is not defined
Please try again
Enter the first number: 10
Enter the second number: 2
x / y is 5
```

8.9 最后……

最后，是 `finally` 子句。它可以用来在可能的异常后进行清理。它和 `try` 子句联合使用：

```
x = None
try:
    x = 1 / 0
finally:
    print "Cleaning up..."
    del x
```

上面的代码中，`finally` 子句肯定会被执行，不管 `try` 子句中是否发生异常(在 `try` 子句之前初始化 `x` 的原因是如果不这样做，由于 `ZeroDivisionError` 的存在，`x` 就永远不会被赋值。这样就会导致在 `finally` 子句中使用 `del` 删除它的时候产生异常，而且这个异常是无法捕捉的)。

运行这段代码，在程序崩溃之前，对于变量 `x` 的清理就完成了：

```
Cleaning up...
  File "/home/marlowes/MyPython/My_Exception.py", line 36, in <module> x = 1 / 0
ZeroDivisionError: integer division or modulo by zero
```

注：在*Python2.5*之前的版本内，`finally`子句需要独立使用，而不能作为`try`语句的`except`子句使用。如果都要使用的话，那么需要两条语句。但在*Python2.5*及其之后的版本中，可以尽情地组合这些子句。

8.10 异常和函数

异常和函数能很自然地一起工作。如果异常在函数内引发而不被处理，它就会传播至(浮到)函数调用的地方。如果在那里也没有处理异常，它就会继续传播，一直到达主程序(全局作用域)。如果那里没有异常处理程序，程序会带着栈跟踪中止。看个例子：

```
>>> def faulty():
...     raise Exception("Something is wrong")
...
>>> def ignore_exception():
...     faulty()
...
>>> def handle_exception():
...     try:
...         faulty()
...     except:
...         print "Exception handled"
...
>>> ignore_exception()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module> File "<stdin>", line 2, in ignore_exception
  File "<stdin>", line 2, in faulty
Exception: Something is wrong
>>> handle_exception()
Exception handled
```

可以看到，`faulty`中产生的异常通过`faulty`和`ignore_exception`传播，最终导致了栈跟踪。同样地，它也传播到了`handle_exception`，但在这个函数中被`try/except`语句处理。

8.11 异常之禅

异常处理并不是很复杂。如果知道某段代码可能会导致某种异常，而又不希望程序以堆栈跟踪的形式终止，那么就根据需要添加`try/except`或者`try/finally`语句(或者它们的组合)进行处理。

有些时候，条件语句可以实现和异常处理同样的功能，但是条件语句可能在自然性和可读性上差些。而从另一方面来看，某些程序中使用`if/else`实现会比使用`try/except`要好。让我们看几个例子。

假设有一个字典，我们希望打印出存储在特定的键下面的值。如果该键不存在，那么什么也不做。代码可能像下面这样写：

```
def describePerson(person):
    print "Description of", person["name"]
    print "Age:", person["age"]
    if "occupation" in person:
        print "Occupation:", person["occupation"]
```

如果给程序提供包含名字 `Throatwobbler Mangrove` 和年龄 `42` (没有职业)的字典的函数，会得到如下输出：

```
Description of Throatwobbler Mangrove
Age: 42
```

如果添加了职业 `camper` ，会的到如下输出：

```
Description of Throatwobbler Mangrove
Age: 42
Occupation: camper
```

代码非常直观，但是效率不高(尽管这里主要关心的是代码的简洁性)。程序会两次查找 `"occupation"` 键，其中一次用来检查键是否存在(在条件语句中)，另外一次获得值(打印)。另外一个解决方案如下：

```
def describePerson(person):
    print "Description of", person["name"]
    print "Age:", person["age"]
    try:
        print "Occupation: " + person["occupation"]
    except KeyError:
        pass
```

注：这里在打印职业时使用加号而不是逗号。否则字符串 *`"Occupation:"`* 在异常引发之前就会被输出。

这个程序直接假定 `"occupation"` 键存在。如果它的确存在，那么就会省事一些。直接取出它的值再打印输出即可——不用额外检查它是否真的存在。如果该键不存在，则会引发 `KeyError` 异常，而被 `except` 子句捕捉到。

在查看对象是否存在特定特性时， `try/except` 也很有用。假设想要查看某对象是否有 `write` 特性，那么可以使用如下代码：

```
try:
    obj.write
except AttributeError:
    print "The object is not writeable"
else:
    print "The object is writeable"
```

这里的 `try` 子句仅仅访问特性而不用对它做别的有用的事情。如果 `AttributeError` 异常引发，就证明对象没有这个特性，反之存在该特性。这是实现第七章中介绍的 `getattr` (7.2.8节)方法的替代方法，至于更喜欢哪种方法，完全是个人喜好。其实在内部实现 `getattr` 时也是使用这种方法：它试着访问特性并且捕捉可能引发的 `AttributeError` 异常。

注意，这里所获得的效率提高并不多(微乎其微)，一般来说(除非程序有性能问题)程序开发人员不用过多担心这类优化问题。在很多情况下，使用 `try/except` 语句比使用 `if/else` 会更自然一些(更“Python化”)，应该养成尽可能使用 `try/except` 语句的习惯。

8.12 小结

本章的主题如下。

☑ 异常对象：异常情况(比如发生错误)可以用异常对象表示。它们可以用几种方法处理，但是如果忽略的话，程序就会中止。

☑ 警告：警告类似于异常，但是(一般来说)仅仅打印错误信息。

☑ 引发异常：可以使用 `raise` 语句引发异常。它接受异常类或者异常实例作为参数。还能提供两个参数(异常和错误信息)。如果在except子句中不使用参数调用 `raise` ，它就会“重新引发”该子句捕捉到的异常。

☑ 自定义异常类：用继承 `Exception` 类的方法可以创建自己的异常类。

☑ 捕捉异常：使用 `try` 语句的 `except` 子句捕捉异常。如果在 `except` 子句中不特别指定异常类，那么所有的异常都会被捕捉。异常可以放在元组中以实现多个异常的指定。如果给 `except` 提供两个参数，第二个参数就会绑定到异常对象上。同样，在一个 `try/except` 语句中能包含多个 `except` 子句，用来分别处理不同的异常。

☑ `else` 子句：除了 `except` 子句，可以使用 `else` 子句。如果主 `try` 块中没有引发异常， `else` 子句就会被执行。

☑ `finally` ：如果需要确保某些代码不管是否有异常引发都要执行(比如清理代码)，那么这些代码可以放置在 `finally` (注意，在Python2.5以前，在一个 `try` 语句中不能同时使用 `except` 和 `finally` 子句——但是一个子句可以放置在另一个子句中)子句中。

☑ 异常和函数：在函数内引发异常时，它就会被传播到函数调用的地方(对于方法也是一样)。

8.12.1 本章的新函数

本章涉及的新函数如表8-2所示。

表8-2 本章的新函数

```
warnings,filterwarnings(action, ...)  用于过滤警告
```

8.12.2 接下来学什么

本章讲异常，内容可能有些意外(双关语)，而下一章的内容真的很不可思议，恩，近乎不可思议。

第九章 魔法方法、属性和迭代器

> 来源：http://www.cnblogs.com/Marlowes/p/5437223.html
>
> 作者：Marlowes

在Python中，有的名称会在前面和后面都加上两个下划线，这种写法很特别。前面几章中已经出现过一些这样的名称(如 `__future__`)，这种拼写表示名字有特殊含义，所以绝不要在自己的程序中使用这样的名字。在Python中，由这些名字组成的集合所包含的方法称为魔法(或特殊)方法。如果对象实现了这些方法中的某一个，那么这个方法会在特殊的情况下(确切地说是根据名字)被Python调用。而几乎没有直接调用它们的必要。

本章会详细讨论一些重要的魔法方法(最重要的是 `__init__` 方法和一些处理对象访问的方法，这些方法允许你创建自己的序列或者映射)。本章还会讲两个相关的主题：属性(在以前版本的Python中通过魔法方法来处理，现在通过 `property` 函数)和迭代器(使用魔法方法 `__iter__` 来允许迭代器在 `for` 循环中使用)，本章最后还有一个相关的示例，其中有些知识点已经介绍过，可以用于处理一些相对复杂的问题。

9.1 准备工作

很久以前(Python2.2中)，对象的工作方式就有了很大的改变。这种改变产生了一些影响，对于刚开始使用Python的人来说，大多数改变都不是很重要(在Alex Martelli所著的《Python技术手册》(*Python in a Nutshell*)的第八章有关与旧式和新式类之间区别的深入讨论)。值得注意的是，尽管可能使用的是新版的Python，但一些特性(比如属性和 `super` 函数)不会在旧式的类上起作用。为了确保类是新式的，应该把赋值语句 `__metaclass__ = type` 放在你的模块的最开始(第七章提到过)，或者(直接或者间接)子类化内建类(实际上是类型) `object` (或其他一些新式类)。考虑下面的两个类。

```
class NewStyle(object):
    more_code_here
class OldStyle:
    more_code_here
```

在这两个类中， `NewStyle` 是新式的类， `OldStyle` 是旧式的类。如果文件以 `__metaclass__ = type` 开始，那么两个类都是新式类。

注：除此之外，还可以在自己的类的作用域中对 *`__metaclass__`* 变量赋值。这样只会为这个类设定元类。元类就是其他类(或类型)的类——这是个更高及的主题。要了解关于元类的更多信息，请参见*Guido van Rossum*的技术性文章*Unifying types and classes in Python 2.2*。或者在互联网上搜索术语*python metaclasses*。

在本书所有示例代码中都没有显式地设定元类(或者子类化`object`)。如果要没有兼容之前旧版本Python的需要，建议你将所有类写为新式的类，并使用`super`函数(稍后会在9.2.3节讨论)这样的特性。

注：在*Python3.0*中没有“旧式”的类，也不需要显式地子类化*`object`*或者将元类设为*`type`*。所有的类都会隐式地成为*`object`*的子类——如果没有明确超类的话，就会直接子类化；否则会间接子类化。

9.2 构造方法

首先要讨论的第一个魔法方法是构造方法。如果读者以前没有听过构造方法这个词，那么说明一下：构造方法是一个很奇特的名字，它代表着类似于以前例子中使用过的那种名为`init`的初始化方法。但构造方法和其他普通方法不同的地方在于，当一个对象被创建后，会立即调用构造方法。因此，刚才我做的那些工作现在就不用做了：

```
>>> f = FooBar()
>>> f.init()
# 构造方法能让它简化成如下形式:
>>> f = FooBar()
```

在Python中创建一个构造方法很容易。只要把`init`方法的名字从简单的`init`修改为魔法版本`__init__`即可：

```
>>> class FooBar:
... def __init__(self):
...         self.somevar = 19
...
>>> f = FooBar()
>>> f.somevar
19
```

现在一切都很好。但如果给构造方法传几个参数的话，会有什么情况发生呢？看看下面的代码：

```
class FooBar:
    def __init__(self, value=19):
        self.somevar = value
```

你认为可以怎样使用它呢？因为参数是可选的，所以你可以继续，，当什么事也没发生。但如果要使用参数(或者不让参数是可选的)的时候会发生什么？我相信你已经猜到了，一起来看看结果吧：

```
>>> f = FooBar("This is a constructor argument")
>>> f.somevar 'This is a constructor argument'
```

在Python所有的魔法方法中，`__init__`是使用最多的一个。

注：*Python*中有一个魔法方法叫做`__del__`，也就是析构方法。它在对象就要被垃圾回收之前调用。但发生调用的具体时间是不可知的。所以建议读者尽力避免使用`__del__`函数。

9.2.1 重写一般方法和特殊的构造方法

第七章中介绍了继承的知识。每个类都可能拥有一个或多个超类，它们从超类那里继承行为方式。如果一个方法在B类的一个实例中被调用(或一个属性被访问)，但在B类中没有找到该方法，那么就会去它的超类A里面找。考虑下面的两个类：

```
class A:
    def Hello(self):
        print "Hello, I'm A."

class B(A):
    pass
```

A类定义了一个叫做`Hello`的方法，被B类继承。下面是一个说明类是如何工作的例子：

```
>>> a = A()
>>> b = B()
>>> a.hello()
Hello, I'm A.
>>> b.hello()
Hello, I'm A.
```

因为B类没有定义自己的`Hello`方法，所以当`hello`被调用时，原始的信息就被打印出来。

在子类中增加功能的最基本的方法就是增加方法。但是也可以重写一些超类的方法来自定义继承的行为。B类也能重写这个方法。比如下面的例子中B类的定义就被修改了。

```
class B(A):
    def hello(self):
        print "Hello, I'm B."
```

使用这个定义，`b.hello()`能产生一个不同的结果。

```
>>> b = B()
>>> b.hello()
Hello, I'm B.
```

重写是继承机制中的一个重要内容，对于构造方法尤其重要。构造方法用来初始化新创建对象的状态，大多数子类不仅要拥有自己的初始化代码，还要拥有超类的初始化代码。虽然重写的机制对于所有方法来说都是一样的，但是当处理构造方法比重写方法时，更可能遇到特别的问题：如果一个类的构造方法被重写，那么就需要调用超类(你所继承的类)的构造方法，否则对象可能不会被正确地初始化。

考虑下面的`Bird`类：

```
class Bird:
    def __init__(self):
        self.hungry = True
    def eat(self):
        if self.hungry:
            print "Aaaah..."
            self.hungry = False
        else:
            print "No, thanks!"
```

这个类定义所有的鸟都具有的一些最基本的能力：吃。下面就是这个类的用法示例：

```
>>> b = Bird()
>>> b.eat()
Aaaah...
>>> b.eat()
No, thanks!
```

就像能在这个例子中看到的，鸟吃过了以后，它就不再饥饿。现在考虑子类 `SongBird` ，它添加了唱歌的行为：

```
class SongBird(Bird):
    def __init__(self):
        self.sound = "Squawk!"

    def sing(self):
        print self.sound
```

`SongBird` 类和 `Bird` 类一样容易使用：

```
>>> sb = SongBird()
>>> sb.sing()
Squawk!
```

因为 `SongBird` 和 `Bird` 的一个子类，它继承了 `eat` 方法，但如果调用 `eat` 方法，就会产生一个问题：

```
>>> sb.eat()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "Magic_Methods.py", line 27, in eat if self.hungry:
AttributeError: SongBird instance has no attribute 'hungry'
```

异常很清楚地说明了错误： `SongBIrd` 没有 `hungry` 特性。原因是这样的：在 `SongBird` 中，构造方法被重写，但新的构造方法没有任何关于初始化 `hungry` 特性的代码。为了达到预期的效果， `SongBird` 的构造方法必须调用其超类 `Bird` 的构造方法来确保进行基本的初始化。有两种方法能达到这个目的：调用超类构造方法的未绑定版本，或者使用super函数。后面两节会介绍这两种技术。

9.2.2 调用未绑定的超类构造方法

本节所介绍的内容只要是历史遗留问题。在目前版本的Python中，使用 `super` 函数(下一节会介绍)会更为简单明了(在Python3.0中更是如此)。但是很多遗留的代码还会使用本节介绍的方法，所以读者还是要了解一些。而且它很有用，这是一个了解绑定和未绑定方法之间区别的好例子。

那么下面进入实际内容。不要被本节的标题吓到，放松。其实调用超类的构造方法很容易(也很有用)。下面我先给出在上一节末尾提出的问题的解决方法。

```
class SongBird(Bird):
    def __init__(self):
        Bird.__init__(self)
        self.sound = "Squawk!"

    def sing(self):
        print self.sound
```

`SongBird` 类中至添加了一行代码—— `Bird.__init__(self)` 。在解释这句话真正的含义之前，首先来演示一下执行效果：

```
>>> sb = SongBird()
>>> sb.sing()
Squawk!
>>> sb.eat()
Aaaah...
>>> sb.eat()
No, thanks!
```

为什么会有这样的结果？在调用一个实例的方法时，该方法的 `self` 参数会被自动绑定到实例上(这称为绑定方法)。前面已经给出几个类似的例子了。但是如果直接调用类的方法(比如 `Bird.__init__`)，那么就没有实例会被绑定。这样就可以自由地提供需要的 `self` 参数。这样的方法称为未绑定(unbound)方法，也就是这节标题中所提到的。

通过将当前的实例作为 `self` 参数提供给未绑定方法， `SongBird` 就能够使用其超类构造方法的所有实现，也就是说属性 `hungry` 能被设置。

9.2.3 使用 `super` 函数

如果读者不想坚守旧版本的Python阵营，那么就应该使用 `super` 函数。它只能在新式类中使用，不管怎么样，你都应该尽量使用新式类。当前的类和对象可以作为 `super` 函数的参数使用，调用函数返回的对象的任何方法都是调用超类的方法，而不是当前类的方法。那么就可以不用在SongBird构造方法中使用 `Bird` ，而直接调用 `super(SongBird, self)` 。除此之外， `__init__` 方法能以一个普通的(绑定)方式被调用。

注：*在Python3.0中， `super` 函数可以不带任何参数进行调用，功能依然具有“魔力”。*

下面的例子是对 `Bird` (注意 `Bird` 是 `object` 的子类，这样它就成为了一个新式类)例子的更新：

```
# super函数只在新式类中起作用
__metaclass__ = type

class Bird:
    def __init__(self):
        self.hungry = True
    def eat(self):
        if self.hungry:
            print "Aaaah..."
            self.hungry = False
        else:
            print "No, thanks!"

class SongBird(Bird):
    def __init__(self):
        super(SongBird, self).__init__()
        self.sound = "Squawk!"

    def sing(self):
        print self.sound
```

这个新式的版本的运行结果和旧式版本的一样：

```
>>> sb = SongBird()
>>> sb.sing()
Squawk!
>>> sb.eat()
Aaaah...
>>> sb.eat()
No, thanks!
```

为什么 `super` 函数这么超级

在我看来， `super` 函数比在超类中直接调用未绑定方法更直观。但这并不是它的唯一优点。 `super` 函数实际上是很智能的，因此即使类已经继承多个超类，它也只需要使用一次 `super` 函数(但要确保所有的超类的构造方法都使用了 `super` 函数)。在一些含糊的情况下使用旧式类会很别扭(比如两个超类共同继承一个超类)，但能被新式类和 `super` 函数自动处理。内部的具体工作原理不用理解，但必须清楚地知道：在大多数情况下，使用新式类和super函数是比调用超类的未绑定的构造方法(或者其他的方法)更好的选择。

那么， `super` 函数到底返回什么？一般来说读者不用担心这个问题，就假装它返回的所需的超类好了。实际上它返回了一个 `super` 对象，这个对象负责进行方法解析。当对其特性进行访问时，它会查找所有的超类(以及超类的超类)，直到找到所需的特性为止(或者引发一个 `AttributeError` 异常)。

9.3 成员访问

尽管 `__init__` 是目前为止提到的最重要的特殊方法，但还有一些其他的方法提供的作用也很重要。本节会讨论常见的魔法方法的集合，它可以创建行为类似于序列或映射的对象。

基本的序列和映射的规则很简单，但如果要实现它们全部功能就需要实现很多魔法函数。幸好，还是有一些捷径的，下面马上会说到。

注：规则*(protocol)*这个词在*Python*中会经常使用，用来描述管理某种形式的行为的规则。这与第七章中提到的接口的概念有点类似。规则说明了应该实现何种方法和这些方法应该做什么。因为*Python*中的多态性是基于对象的行为的*(*而不是基于祖先，例如它所属的类或超类，等等*)*。这是一个重要的概念：在其他的语言中对象可能被要求属于某一个类，或者被要求实现某个接口，但*Python*中只是简单地要求它遵循几个给定的规则。因此成为了一个序列，你所需要做的只是遵循序列的规则。

9.3.1 基本的序列和映射规则

序列和映射是对象的集合。为了实现它们基本的行为(规则)，如果对象是不可变的，那么就需要使用两个魔法方法，如果是可变的则需要四个。

☑ `__len__(self)` ：这个方法应该返回级和中所含项目的数量。对于序列来说，这就是元素的个数；对于映射来说，则是键-值对的数量。如果 `__len__` 返回0(并且没有实现重写该行为的 `__nozero__`)，对象会被当作一个布尔变量中的假值(空的列表、元组、字符串和字典也一样)进行处理。

☑ `__getitem__(self, key)` ：这个方法返回与所给的键对应的值。对于一个序列，键应该是一个0～n-1的整数(或者像后面所说的负数)，n是序列的长度；对于映射来说，可以使用任何种类的键。

☑ `__setitem__(self, key, value)` ：这个方法应该按一定的方式存储和key相关的value，该值随后可使用 `__getitem__` 来获取。当然，只能为可以修改的对象定义这个方法。

☑ `__delitem__(self, key)` ：这个方法在对一部分对象使用del语句时被调用，同时必须删除和键相关的键。这个方法也是为可修改的对象定义的(并不是删除全部的对象，而只删除一些需要移除的元素)。

对于这些方法的附加要求如下。

☑ 对于一个序列来说，如果键是负整数，那么要从末尾开始计数。换句话说就是 `x[-n]` 和 `x[len(x)-n]` 是一样的。

☑ 如果键是不合适的类型(例如，对序列使用字符串作为键)，会引发一个 `TypeError` 异常。

☑ 如果序列的索引是正确的类型，但超出了范围，应该引发一个 `IndexError` 异常。

让我们实践一下，看看如果创建一个无穷序列，会发生什么：

```
__metaclass__ = type
def checkindex(key):
    """ 所给的键是能接受的索引吗？
        为了能被接受，键应该是一个非负的整数。如果它不是一个整数，会引发TypeError；
        如果它是负数，则会引发IndexError(因为序列是无限长的)。 """

    if not isinstance(key, (int, long)):
        raise TypeError
    if key < 0:
        raise IndexError
class ArithmeticSequence:
    def __init__(self, start=0, step=1):
        """ 初始化算术序列
        起始值——序列中的第一个值
        步长——两个相邻值之间的差别
        改变——用户修改的值的字典 """
        self.start = start  # 保存开始值
        self.step = step  # 保存步长值
        self.changed = {}  # 没有项被修改

    def __getitem__(self, key):
        """ Get an item from the arithmetic sequence. """
        checkindex(key)
        try:
            # 修改了吗？
            return self.changed[key]
        except KeyError:
            # 否则······
            # ······计算值
            return self.start + key * self.step

    def __setitem__(self, key, value):
        """ 修改算术序列中的一个项 """
        checkindex(key)
        # 保存更改后的值
        self.changed[key] = value
```

这里实现的是一个算术序列，该序列中的每个元素都比它前面的元素大一个常数。第一个值是由构造方法参数 `start` (默认为0)给出的，而值与值之间的步长是由 `step` 设定的(默认为1). 用户能将特例规则保存在名为 `changed` 的字典中，从而修改一些元素的值，如果元素没有被修改，那就计算 `self.start + key * self.steo` 的值。

下面是如何使用这个类的例子：

```
>>> s = ArithmeticSequence(1, 2)
>>> s[4] 9
>>> s[4] = 2
>>> s[4] 2
>>> s[5] 11
```

注意，没有实现 `__del__` 方法的原因是我希望删除元素是非法的：

```
>>> del s[4]
Traceback (most recent call last):
  File "ArithmeticSequence.py", line 66, in <module>
    del s[4]
AttributeError: __delitem__
```

这个类没有 `__len__` 方法，因为它是无限长的。

如果使用了一个非法类型的索引，就会引发 `TypeError` 异常，如果索引的类型是正确的但超出了范围(在本例中为负数)，则会引发 `IndexError` 异常：

```
>>> s["four"]
Traceback (most recent call last):
  File "/home/marlowes/Program/Py_Project/ArithmeticSequence.py", line 66, in <module>
 s["four"]
  File "/home/marlowes/Program/Py_Project/ArithmeticSequence.py", line 44, in __getite
m__ checkindex(key)
  File "/home/marlowes/Program/Py_Project/ArithmeticSequence.py", line 18, in checkind
ex raise TypeError
TypeError
>>> s[-42]
Traceback (most recent call last):
  File "/home/marlowes/Program/Py_Project/ArithmeticSequence.py", line 66, in <module>
 s[-42]
  File "/home/marlowes/Program/Py_Project/ArithmeticSequence.py", line 44, in __getite
m__ checkindex(key)
  File "/home/marlowes/Program/Py_Project/ArithmeticSequence.py", line 21, in checkind
ex raise IndexError
IndexError
```

索引检查是通过用户自定义的 `checkindex` 函数实现的。

有一件关于 `checkindex` 函数的事情可能会让人吃惊，即 `isinstance` 函数的使用(这个函数应尽量避免使用，因为类或类型检查和Python中多态的目标背道而驰)。因为Python的语言规范上明确指出索引必须是整数(包括长整数)，所以上面的代码才会如此使用。遵守标准是使用类型检查的(很少的)正当理由之一。

注：分片操作也是可以模拟的。当对支持 `__getitem__` 方法的实例进行分片操作时，分片对象作为键提供。分片对象在Python库参考的2.1节中slice函数部分有介绍。Python2.5有一个更加专门的方法叫做 `__index__` ，它允许你在分片中使用非整形限制。只要你想处理基本序列规则之外的事情，那么这个方法尤其有用。

9.3.2 子类化列表，字典和字符串

到目前为止本书已经介绍了基本的序列/映射规则的4个方法，官方语言规范也推荐实现其他的特殊方法和普通方法(参见[Python参考手册的3.4.5节]
http://www.python.org/doc/ref/sequence-types.html)，包括9.6节描述的`__iter\__`方法在内。实现所有的这些方法(为了让自己的对象和列表或者字典一样具有多态性)是一项繁重的工作，并且很难做好。如果只想在一个操作中自定义行为，那么其他的方法就不用实现。这就是程序员的懒惰(也是常识)。

那么应该怎么做呢？关键词是继承。能继承的时候为什么还要全部实现呢？标准库有3个关于序列和映射规则(`UserList` 、 `UserString` 和 `UserDict`)可以立即使用的实现，在较新版本的Python中，可以子类化内建类型(注意，如果类的行为和默认的行为很接近这就很有用，如果需要重新实现大部分方法，那么还不如重新写一个类)。

因此，如果希望实现一个和内建列表行为相似的序列，可以子类化 `list` 来实现。

注：当子类化一个内建类型——比如 *list* 的时候，也就间接地将 *object* 子类化了。因此该类就自动成为新式类，这就意味着可以使用像 *super* 函数这样的特性了。

下面看看例子，带有访问计数的列表。

```
__metaclass__ = type
class CounterList(list):
    def __init__(self, *args):
        super(CounterList, self).__init__(*args)
        self.counter = 0
    def __getitem__(self, index):
        self.counter += 1
        return super(CounterList, self).__getitem__(index)
```

`CounterList` 类严重依赖于它的超类(`list`)的行为。 `CounterList` 类没有重写的任何方法(和 `append` 、 `extend` 、 `index` 一样)都能被直接使用。在两个被重写的方法中， `super` 方法被用来调用相应的超类的方法，只在 `__init__` 中添加了所需的初始化 `counter` 特性的行为，并在 `__getitem__` 中更新了 `counter` 特性。

注：重写 *__getitem__* 并非获取用户访问的万全之策，因为还有其他访问列表内容的途径，比如通过 *pop* 方法。

这里是 `CounterList` 如何使用的例子：

```
>>> cl = CounterList(range(10))
>>> cl
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> cl.reverse()
>>> cl
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
>>> del cl[3:6]
>>> cl
[9, 8, 7, 3, 2, 1, 0]
>>> cl.counter
0
>>> cl[4] + cl[2]
9
>>> cl.counter
2
```

正如看到的， `CounterList` 在很多方面和列表的作用一样，但它有一个 `counter` 特性(被初始化为0)，每次列表元素被访问时，它都会自增，所以在执行加法 `cl[4] + cl[2]` 后，这个值自增两次，变为2。

9.4 更多魔力

魔法名称的用途有很多，我目前所演示的只是所有用途中的一小部分。大部分的特殊方法都是为高级的用法准备的，所有我不会在这里详细讨论。单是如果感兴趣，可以模拟数字，让对象像函数那样被调用，影响对象的比较，等等。关于特殊函数的更多内容请参考《Python参考手册》中的3.4节。

9.5 属性

第七章曾经提到过访问器方法。访问器是一个简单的方法，它能够使用`getHeight`、`setHeight`这样的名字来得到或者重绑定一些特性(可能是类的私有属性——具体内容请参见第七章的“再论私有化”的部分)。如果在访问给定的特性时必须要采取一些行动，那么像这样的封装状态变量(特性)就很重要。比如下面例子中的`Rectangle`类：

```
__metaclass__ = type
class Rectangle:
    def __init__(self):
        self.width = 0
        self.height = 0
    def setSize(self, size):
        self.width, self.height = size
    def getSize(self):
        return self.width, self.height
```

下面的例子演示如何使用这个类：

```
>>> r = Rectangle()
>>> r.width = 10
>>> r.height = 5
>>> r.getSize()
(10, 5)
>>> r.setSize((150, 100))
>>> r.width 150
```

在上面的例子中，`getSize`和`setSize`方法是一个名为`size`的假想特性的访问器方法，`size`是由`width`和`height`构成的元组。读者可以随意使用一些更有趣的方法替换这里的函数，例如计算面积或者对角线的长度。这些代码没错，但却有缺陷。当程序员使用这个类时不应该还要考虑它是怎么实现的(封装)。如果有一天要改变类的实现，将`size`变成一个真正的特性，这样`width`和`height`就可以动态算出，那么就要把它们放到一个访问器方法中去，并且任何使用这个类的程序都必须重写。客户代码(使用代码的代码)应该能够用同样的方式对待所有特性。

那么怎么解决呢？把所有的属性都放到访问器方法中？这当然没问题。但如果有很多简单的特性，那么就很不现实了(有点笨)。如果那样做就得写很多访问器方法，它们除了返回或者设置特性就不做任何事了。复制加粘贴式或切割代码式的编程方式显然是很糟糕的(尽管是在一些语言中针对这样的特殊问题很普遍)。幸好，Python能隐藏访问器方法，让所有特性看起来一样。这些通过访问器定义的特性被称为属性。

实际上在Python中有两种创建属性的机制。我主要讨论新的机制——只在新式类中使用的property函数，然后我会简单地说明一下如何使用特殊方法实现属性。

9.5.1 `property` 函数

property函数的使用很简单。如果已经编写了一个像上节的`Rectangle`那样的类，那么只要增加一行代码(除了要子类化`object`，或者使用`__metaclass__ = type`语句以外)：

```
__metaclass__ = type
class Rectangle:
    def __init__(self):
        self.width = 0
        self.height = 0
    def setSize(self, size):
        self.width, self.height = size
    def getSize(self):
        return self.width, self.height

    size = property(getSize, setSize)
```

在这个新版的`Rectangle`中，`property`函数创建了一个属性，其中访问器函数被用作参数(先是取值，然后是赋值)，这个属性命名为`size`。这样一来就不再需要担心是怎么实现的了，可以用同样的方式处理`width`、`height`和`size`。

```
>>> r = Rectangle()
>>> r.width = 10
>>> r.height = 5
>>> r.size
(10, 5)
>>> r.size = 150, 100
>>> r.width
150
```

很明显，`size`特性仍然取决于`getSize`和`setSize`中的计算。但它们看起来就像普通的属性一样。

注：如果属性的行为很奇怪，那么要确保你所使用的类为新式类(通过直接或间接子类化`object`，或直接设置元类)；如果不是的话，虽然属性的取值部分还是可以工作，但赋值部分就不一定了(取决于Python的版本)，这很让人困惑。

实际上，`property`函数可以用0、1、3或者4个参数来调用。如果没有参数，产生的属性既不可读，也不可写。如果只使用一个参数调用(一个取值方法)，产生的属性是只读的。第3个参数(可选)是一个用于删除特性的方法(它不要参数)。第4个参数(可选)是一个文档字符串。`property`的四个参数分别被叫做`fget`、`fset`、`fdel`和`doc`，如果想要第一个属性是只写的，并且有一个文档字符串，可以使用关键字参数的方式来实现。

尽管这一节很短(只是对`property`函数的简单说明)，但它却十分的重要。理论上说，在新式类中应该使用`property`函数而不是访问器方法。

它是如何工作的

有的读者很想知道`property`函数是如何实现它的功能的，那么在这里解释一下，不感兴趣的读者可以跳过。实际上，`property`函数不是一个真正的函数，它是其实例拥有很多特殊方法的类，也正是那些方法完成了所有的工作。涉及的方法是`__get__`、`__set__`和`__delete__`。这3个方法和在一起，就定义了描述符规则。实现了

其中任何一个方法的对象就叫描述符(descriptor)。描述符的特殊之处在于它们使如何被访问的。比如，程序读取一个特性时(尤其是在实例中访问该特性，但该特性在类中定义时)，如果该特性被绑定到了实现了 `__get__` 方法的对象上，那么就会调用 `__get__` 方法(结果值也会被返回)，而不只是简单地返回对象。实际上这就是属性的机制，即绑定方法，静态方法和类成员方法(下一节会介绍更多的信息)还有 `super` 函数。《Python参考手册》包括有关描述符规则的简单说明。一个更全面的信息源是Raymond Hettinger的How To Guide for Descriptors

9.5.2 静态方法和类成员方法

在讨论实现属性的旧方法前，先让我们绕道而行，看看另一对实现方法和新式属性的实现方法类似的特征。静态方法和类成员方法分别在创建时分别被装入 `staticmethod` 类型和 `classmethod` 类型对象中。静态方法的定义没有 `self` 参数，且能够被类本身直接调用。类方法在定义时需要名为 `cls` 的类似于 `self` 的参数，类成员方法可以直接用类的具体对象调用。但 `cls` 参数是自动被绑定到类的，请看下面的例子：

```
__metaclass__ = type
class MyClass:
    def smeth():
        print "This is a static method"
    smeth = staticmethod(smeth)
    def cmeth(cls):
        print "This is a class method of", cls
    cmeth = classmethod(cmeth)
```

手动包装和替换方法的技术看起来有些单调，在Python2.4中，为这样的包装方法引入了一个叫做装饰器(decorator)的新语法(它能对任何可调用的对象进行包装，既能够用于方法也能用于函数)。使用 `@` 操作符，在方法(或函数)的上方将装饰器列出，从而指定一个或者更多的装饰器(多个装饰器在应用时的顺序与指定顺序相反)。

```
__metaclass__ = type class MyClass:

    @staticmethod
    def smeth():
        print "This is a static method"
    @classmethod
    def cmeth(cls):
        print "This is a class method of", cls
```

定义了这些方法以后，就可以想下面的例子那样使用(例子中没有实例化类)：

```
>>> MyClass.smeth()
This is a static method
>>> MyClass.cmeth()
This is a class method of <class '__main__.MyClass'>
```

静态方法和类成员方法在Python中向来都不是很重要，主要原因是大部分情况下可以使用函数或者绑定方法代替。在早期的版本中没有得到支持也是一个原因。但即使看不到两者在当前代码中的大量应用，也不要忽视静态方法和类成员方法的应用(比如工厂函数)，可以好好地

考虑一下使用新技术。

9.5.3 `__getattr__` 、 `__setattr__` 和它的朋友们

拦截(intercept)对象的所有特性访问是可能的，这样可以用旧式类实现属性(因为 `property` 方法不能使用)。为了在访问特性的时候可以执行代码。必须使用一些魔法方法。下面的4种方法提供了需要的功能(在旧式类中只需要后3个)。

☑ `__getattribute__(self, name)` ：当特性 `name` 被访问时自动被调用(只能在新式类中使用)。

☑ `__getattr__(self, name)` ：当特性 `name` 被访问且对象没有相应的特性时被自动调用。

☑ `__setattr__(self, name, value)` ：当试图给特性 `name` 赋值时会被自动调用。

☑ `__delatr__(self, name)` ：当试图删除特性 `name` 时被自动调用。

尽管和使用 `property` 函数相比有点复杂(而且在某些方面效率更高)，单这些特殊方法是很强大的，因为可以对处理很多属性的方法进行再编码。

下面还是 `Rectangle` 的例子，但这次使用的是特殊方法：

```
__metaclass__ = type
class Rectangle:
    def __init__(self):
        self.width = 0
        self.height = 0
    def __setattr__(self, name, value):
        if name == "size":
            self.width, self.height = value
        else:
            self.__dict__[name] = value
    def __getattr__(self, name):
        if name == "size":
            return self.width, self.height
        else:
            raise AttritubeError
```

这个版本的类需要注意增加的管理细节。当思考这个例子时，下面的两点应该引起读者的重视。

☑ `__setattr__` 方法在所涉及的特性不是 `size` 时也会被调用。因此，这个方法必须把两方面都考虑进去：如果属性是 `size` ，那么就像前面那样执行操作，否则就要使用特殊方法 `__dict__` ，该特殊方法包含一个字典，字典里面是所有实例的属性。为了避免 `__setattr__` 方法被再次调用(这样会使程序陷入死循环)， `__dict__` 方法被用来代替普通的特性赋值操作。

☑ `__getattr__` 方法只在普通的特性没有被找到的时候调用，这就是说如果给定的名字不是size，这个特性不存在，这个方法会引起一个 `AttritubeError` 异常。如果希望类和 `hasattr` 或者是 `getattr` 这样的内建函数一起正确地工作， `__getattr__` 方法就很重要。如

果使用的是`size`属性，那么就会使用在前面的实现中找到的表达式，

注：就像死循环陷阱和`__setattr__`有关系一样，还有一个陷阱和`__getattribute__`有关系。因为`__getattribute__`拦截所有特性的访问(在新式类中)，也拦截对`__dict__`的访问！访问`__getattribute__`中与`self`相关的特性时，使用超类的`__getattribute__`方法(使用`super`函数)是唯一安全的途径。

9.6 迭代器

在前面的章节中，我提到过迭代器(和可迭代)，本节将对此进行深入讨论。只讨论一个特殊方法——`__iter__`，这个方法是迭代器规则(iterator protocol)的基础。

9.6.1 迭代器规则

迭代的意思是重复做一些事情很多次，就像在循环中做的那样。到现在为止只在for循环中对序列和字典进行过迭代，但实际上也能对其他对象进行迭代：只要该对象实现了`__iter__`方法。

`__iter__`方法会返回一个迭代器(iterator)，所谓的迭代器就是具有`next`方法(这个方法在调用时不需要任何参数)的对象。在调用`next`方法时，迭代器会返回它的下一个值。如果`next`方法被调用，但迭代器没有值可以返回，就会引发一个`StopIteration`异常。

注：迭代器规则在*Python3.0*中有一些变化。在新的规则中，迭代器对象应该实现`__next__`方法，而不是`next`。而新的内建函数`next`可以用于访问这个方法。换句话说，`next(it)`等同于*3.0*之前版本中的`it.next()`。

迭代规则的关键是什么？为什么不使用列表？因为列表的杀伤力太大。如果有一个函数，可以一个接一个地计算值，那么在使用时可能是计算一个值时获取一个值——而不是通过列表一次性获取所有值。如果有很多值，列表就会占用太多的内存。但还有其他的理由：使用迭代器更通用、更简单、更优雅。让我们看看一个不使用列表的例子，因为要用的话，列表的长度必须无限。

这里的“列表”是一个斐波那契数列。使用的迭代器如下：

```
__metaclass__ = type
class Fibs:
    def __init__(self):
        self.a = 0
        self.b = 1

    def next(self):
        self.a, self.b = self.b, self.a + self.b
        return self.a
    def __iter__(self):
        return self
```

注意，迭代器实现了 `__iter__` 方法，这个方法实际上返回迭代器本身。在很多情况下，`__iter__` 会放到其他的会在 `for` 循环中使用的对象中。这样一来，程序就能返回所需的迭代器。此外，推荐使迭代器实现它自己的 `__iter__` 方法，然后就能直接在 `for` 循环中使用迭代器本身了。

注：正式的说法是，一个实现了 *`__iter__`* 方法的对象是可迭代的，一个实现了 *`__next__`* 方法的对象则是迭代器。

首先，产生一个 `Fibs` 对象：

```
>>> fibs = Fibs()
```

可在 `for` 循环中使用该对象——比如去查找在斐波那契数列中比1000大的数中的最小的数：

```
for f in fibs:
    if f > 1000:
        print f break ··· 1597
```

因为设置了 `break` ，所以循环在这里停止了，否则循环会一直继续下去。

注：内建函数 *`list`* 可以从可迭代的对象中获得迭代器。

```
>>> it = iter([1, 2, 3])
>>> it.next()
1
>>> it.next()
2
```

除此之外，它也可以从函数或者其他可调用对象中获取可迭代对象*(请参见Python库参考获取更多信息)*。

9.6.2 从迭代器得到序列

除了在迭代器和可迭代对象上进行迭代(这是经常做的)外，还能把它们转换为序列。在大部分能使用序列的情况下(除了在索引或者分片等操作中)，都能使用迭代器(或者可迭代对象)替换。关于这个的一个很有用的例子是使用 `list` 构造方法显式地将迭代器转化为列表。

```
__metaclass__ = type
class TestIterator:
    value = 0
    def next(self):
        self.value += 1
        if self.value > 10:
            raise StopIteration
        return self.value
    def __iter__(self):
        return self

...
>>> ti = TestIterator()
>>> list(ti)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

9.7 生成器

生成器是Python新引入的概念，由于历史原因，它也叫简单生成器。它和迭代器可能是近几年来引入的最强大的两个特性。但是，生成器的概念则要更高级一些，需要花些功夫才能理解它使如何工作的以及它有什么用处。生成器可以帮助读者写出非常优雅的代码，当然，编写任何程序时不使用生成器也是可以的。

生成器是一种用普通的函数语法定义的迭代器。它的工作方式可以用例子来很好地展现。让我们先看看怎么创建和使用生成器，然后再了解一下它的内部机制。

9.7.1 创建生成器

创建一个生成器就像创建函数一样简单。相信你已经厌倦了斐波那契数列的例子，所以下面会换一个例子来说明生成器的知识。首先创建一个展开嵌套列表的函数。参数是一个列表，和下面这个很像：

```
nested = [[1, 2], [3, 4], [5]]
```

换句话说就是一个列表的列表。函数应该按顺序打印出列表中的数字。解决的办法如下：

```
def flatten(nested):
    for sublist in nested:
        for element in sublist:
            yield element
```

这个函数的大部分是简单的。首先迭代提供的嵌套列表中的所有子列表，然后按顺序迭代子列表中的元素。如果最后一行是 `print element` 的话，那么就容易理解了，不是吗？

这里的 `yield` 语句是新知识。任何包含 `yield` 语句的函数成为生成器。除了名字不同以外，它的行为和普通的函数也有很大的差别。这就在于它不是像 `return` 那样返回值，而是每次产生多个值。每次产生一个值(使用 `yield` 语句)，函数就会被冻结：即函数停在那点等待被重新唤醒。函数被重新唤醒后就从停止的那点开始执行。

接下来可以通过在生成器上迭代来使用所有的值。

```
>>> nested = [[1, 2], [3, 4], [5]]
>>> for num in flatten(nested):
...     print num
1
2
3
4
5

or

>>> list(flatten(nested))
[1, 2, 3, 4, 5]
```

循环生成器

Python2.4引入了列表推导式的概念(请参见第五章)，生成器推导式(或称生成器表达式)和列表推导式的工作方式类似，只不过返回的不是列表而是生成器(并且不会立刻进行循环)。所返回的生成器允许你像下面这样一步一步地进行计算：

```
>>> g = ((i + 2) ** 2 for i in range(2, 27))
>>> g.next() 16
```

和列表推导式不同的就是普通圆括号的使用方式，在这样简单的例子中，还是推荐大家使用列表推导式。但如果读者希望将可迭代对象(例如生成大量的值)“打包”，那么最好不要使用列表推导式，因为它会立即实例化一个列表，从而丧失迭代的优势。

更妙的地方在于生成器推导式可以在当前的圆括号内直接使用，例如在函数调用中，不用增加另一对圆括号，换句话说，可以像下面这样编写代码：

```
sum(i ** 2 for i in range(10))
```

9.7.2 递归生成器

上节创建的生成器只能处理两层嵌套，为了处理嵌套使用了两个 `for` 循环。如果要处理任意层的嵌套该怎么办？例如，可能要使用来表示树形结构(也可用于特定的树类，但原理是一样的)。每层嵌套需要增加一个 `for` 循环，但因为不知道有几层嵌套，所以必须把解决方案变得更灵活。现在是求助于递归(recursion)的时候了。

```
def flatten(nested):
    try:
        for sublist in nested:
            for element in flatten(sublist):
                yield element
    except TypeError:
        yield nested
```

当 `flatten` 被调用时，有两种可能性(处理递归时大部分都是有两种情况)：基本情况和需要递归的情况。在基本情况中，函数被告知展开一个元素(比如一个数字)，这种情况下， `for` 循环会引发一个 `TypeError` 异常(因为试图对一个数字进行迭代)，生成器会产生一个元素。

如果展开的是一个列表(或者其他可迭代对象)，那么就要进行特殊处理。程序必须遍历所有的子列表(一些可能不是列表)，并对它们调用 `flatten` 。然后使用另一个 `for` 循环来产生被展开的子列表中的所有元素。这可能看起来有点不可思议，但却能工作。

```
list(flatten([[[1], 2], 3, 4, [5, [6, 7]], 8]))
[1, 2, 3, 4, 5, 6, 7, 8]
```

这么做只有一个问题：如果 `nested` 是一个类似于字符串的对象(字符串、 `Unicode` 、 `UserString` ，等等)，那么它就是一个序列，不会引发 `TypeError` ，但是你不想对这样的对象进行迭代。

注：不应该在 *`flatten`* 函数中对类似于字符串的对象进行迭代，出于两个主要的原因。首先，需要实现的是将类似于字符串的对象当成原子值，而不是当成应被展开的序列。其次，对它们进行迭代实际上会导致无穷递归，因为一个字符串的第一个元素是另一个长度为*1*的字符串，而长度为*1*的字符串的第一个元素就是字符串本身。

为了处理这种情况，则必须在生成器的开始处添加一个检查语句。试着将传入的对象和一个字符串拼接，看看会不会出现 `TypeError` ，这是检查一个对象是不是类似于字符串的最简单、最快速的方法(感谢Alex Martelli指出了这个习惯用法和在这里使用的重要性)。下面是加入了检查语句的生成器：

```
def flatten(nested):
    try:
        # 不要迭代类似字符串的对象:
        try:
            nested + ""
        except TypeError:
            pass
        else:
            raise TypeError
                for sublist in nested:
                    for element in flatten(sublist):
                        yield element
    except TypeError:
        yield nested
```

如果表达式 `nested + ""` 引发了一个 `TypeError` ，它就会被忽略。然而如果没有引发 `TypeError` ，那么内层 `try` 语句中的 `else` 子句就会引发一个它自己的 `TypeError` 异常。这就会按照原来的样子生成类似于字符串的对象(在 `except` 子句的外面)，了解了吗？

这里有一个例子展示了这个版本的类应用于字符串的情况：

```
>>> list(flatten(["foo", ["bar", ["baz"]]]))
['foo', 'bar', 'baz']
```

上面的代码没有执行类型检查。这里没有测试`nested`是否是一个字符串(可以使用`isinstance`函数完成检查)，而只是检查`nested`的行为是不是像一个字符串(通过和字符串拼接来检查)。

9.7.3 通用生成器

如果到目前的所有例子你都看懂了，那应该或多或少地知道如何使用生成器了。生成器是一个包含`yield`关键字的函数。当它被调用时，在函数体中的代码不会被执行，而会返回一个迭代器。每次请求一个值，就会执行生成器中的代码，直到遇到一个`yield`或者`return`语句。`yield`语句意味着应该生成一个值。`return`语句意味着生成器要停止执行(不再生成任何东西，`return`语句只有在一个生成器中使用时才能进行无参数调用)。

换句话说，生成器是由两部分组成：生成器的函数和生成器的迭代器。生成器的函数是用`def`语句定义的，包含`yield`部分，生成器的迭代器是这个函数返回的部分。按一种不是很准确的说法，两个实体经常被当做一个，合起来叫做生成器。

```
>>> def simple_generator():
...     yield 1
...
>>> simple_generator
<function simple_generator at 0x7fc241cad578>
>>> simple_generator()
<generator object simple_generator at 0x7fc241cf1cd0>
```

生成器的函数返回的迭代器可以像其他的迭代器那样使用。

9.7.4 生成器方法

生成器的新特征(在Python2.5中引入)是在开始运行后为生成器提供值的能力。表现为生成器和“外部世界”进行交流的渠道，要注意下面两点。

☑ 外部作用域访问生成器的`send`方法，就像访问`next`方法一样，只不过前者使用一个参数(要发送的“消息”——任意对象)。

☑ 在内部则挂起生成器，`yield`现在作为表达式而不是语句使用，换句话说，当生成器重新运行的时候，`yield`方法返回一个值，也就是外部通过`send`方法发送的值。如果`next`方法被使用，那么`yield`方法返回`None`。

注意，使用`send`方法(而不是`next`方法)只有在生成器挂起之后才有意义(也就是说在`yield`函数第一次被执行后)。如果在此之前需要给生成器提供更多信息，那么只需使用生成器函数的参数。

注：如果真想对刚刚启动的生成器使用*`send`*方法，那么可以将*`None`*作为其参数进行调用。

下面是一个非常简单的例子，可以说明这种机制：

```
def repeater(value):
    while True:
        new = (yield value)
        if new is not None:
            value = new
```

使用方法如下：

```
>>> r = repeater(42)
>>> r.next()
42
>>> r.send("Hello, world!")
"Hello, world!"
```

注意看 `yield` 表达式周围的圆括号的使用。虽然并未严格要求，但在使用返回值的时候，安全起见还是要闭合 `yield` 表达式。

生成器还有其他两个方法(在Python2.5及以后的版本中)。

☑ `throw` 方法(使用异常类型调用，还有可选的值以及回溯对象)用于在生成器内引发一个异常(在 `yield` 表达式中)。

☑ `close` 方法(调用时不用参数)用于停止生成器。

`close` 方法(在需要的时候也会由Python垃圾收集器调用)也是建立在异常的基础上的。它在 `yield` 运行处引发一个 `GeneratorExit` 异常，所以如果需要在生成器内进行代码清理的话，则可以将 `yield` 语句放在 `try/finally` 语句中。如果需要的话，还可以捕捉 `GeneratorExit` 异常，但随后必须将其重新引发(可能在清理之后)，引发另外一个异常或者直接返回。试着在生成器的 `close` 方法被调用后再通过生成器生成一个值则会导致 `RuntimeError` 异常。

注：有关更多生成器方法的信息，以及如何将生成器转换为简单的协同程序(coroutine)的方法，请参见PEP 342。

9.7.5 模拟生成器

生成器在旧版本的Python中是不可用的。下面介绍的就是如何使用普通的函数模拟生成器。

先从生成器的代码开始。首先将下面语句放在函数体的开始处：

```
result = []
```

如果代码已经使用了 `result` 这个名字，那么应该用其他的名字代替(使用一个更具有描述性的名字是一个好主意)，然后将下面这种形式的代码：

```
yield some_expression
```

用下面的语句替换：

```
result.append(some_expression)
```

最后，在函数的末尾，添加下面这条语句：

```
return result
```

尽管这个版本可能不适用于所有生成器，但对大多数生成器来说是可行的(比如，它不能用于一个无限的生成器，当然不能把它的值放入列表中)。

下面是 `flatten` 生成器用普通的函数重写的版本：

```
def flatten(nested):
    result = []
    try:
        # 不要迭代类似字符串的对象:
        try:
            nested + ""
        except TypeError:
            pass
        else:
            raise TypeError
            for sublist in nested:
                for element in flatten(sublist):
                    result.append(element)
    except TypeError:
        result.append(nested)
    return result
```

9.8 八皇后的问题

现在已经学习了所有的魔法方法，是把它们用于实践的时候了。本节会介绍如何使用生成器解决经典的变成问题。

9.8.1 生成器和回溯

生成器是逐渐产生结果的复杂递归算法的理想实现工具。没有生成器的话，算法就需要一个作为额外参数传递的半成品方案，这样递归调用就可以在这个方案上建立起来。如果使用生成器，那么所有的递归调用只要创建自己的 `yield` 部分。前一个递归版本的 `flatten` 程序中使用的就是后一种做法，相同的策略也可以用在遍历(Traverse)图和树形结构中。

在一些应用程序中，答案必须做很多次选择才能得出。并且程序不只是在一个层面上而必须在递归的每个层面上做出选择。拿生活中的例子打个比方好了，首先想象一下你要出席一个很重要的会议。但你不知道在哪开会，在你的面前有两扇门，开会的地点就在其中一扇门后

面，于是有人挑了左边的进入，然后又发现两扇们。后来再选了左边的门，结果却错了，于是回溯到刚才的两扇门那里，并且选择右边的们，结果还是错的，于是再次回溯，直到回到了开始点，再在那里选择右边的门。

图和树

如果读者没有听过图和树，那么应该尽快学习。它们是程序设计和计算机科学中的重要概念。如果想了解更多，应该找一本与计算机科学、离散数学、数据结构或算法相关的书籍来学习。你可以从下面链接的网页中得到数和图的简单定义：

http://mathworld.wolfram.com/Graph.html

http://mathworld.wolfram.com/Tree.html

http://www.nist.gov/dads/HTML/tree.html

http://www.nist.gov/dads/HTML/graph.html

在互联网上搜索以及访问维基百科全书会获得更多信息。

这样的回溯策略在解决需要尝试每种组合，直到找到一种解决方案的问题时很有用。这类问题能按照下面伪代码的方式解决：

```
# 伪代码
第1层所有的可能性:
    第2层所有的可能性:
        ...
            第n层所有的可能性：
                可行吗?
```

为了直接使用 `for` 循环来实现，就必须知道会遇到的具体判断层数，如果无法得知层数信息，那么可以使用递归。

9.8.2 问题

这是一个深受喜爱的计算机科学谜题：有一个棋盘和8个要放到上面的皇后。唯一的要求是皇后之间不能形成威胁。也就是说，必须把它们放置成每个皇后都不能吃掉其他皇后的状态。怎么样才能做到呢？皇后要如何放置呢？

这是一个典型的回溯问题：首先尝试放置第1个皇后(在第1行)，然后放置第2个，依次类推。如果发现不能放置下一个皇后，就回溯到上一步，试着将皇后放到其他的位置。最后，或者尝试完所有的可能或者找到解决方案。

问题会告知，棋盘上只有八个皇后，但我们假设有任意数目的皇后(这样就更合实际生活中的回溯问题)，怎么解决？如果你要自己解决，那么就不要继续了，因为解决方案马上要给出。

注：实际上对于这个问题有更高效的解决方案，如果想了解更多的细节，那么可以在网上搜索，以得到很多有价值的信息。访问 *http://www.cit.gu.edu.au/~sosic/nqueens.html* 可以找到关于各种解决方案的简单介绍。

9.8.3 状态表示

为了表示一个可能的解决方案(或者方案的一部分)，可以使用元组(或者列表)。每个元组中元素都指示相应行的皇后的位置(也就是列)。如果 `state[0]==3` ，那么就表示在第1行的皇后是在第4列(记得么，我们是从0开始计数的)。当在某一个递归的层面(一个具体的行)时，只能知道上一行皇后的位置，因此需要一个长度小于8的状态元组(或者小于皇后的数目)。

注：使用列表来代替元组表示状态也是可行的。具体使用哪个只是一个习惯的问题。一般来说，如果序列很小而且是静态的，元组是一个好的选择。

9.8.4 寻找冲突

首先从一些简单的抽象开始。为了找到一种没有冲突的设置(没有皇后会被其他的皇后吃掉)，首先必须定义冲突是什么。为什么不在定义的时候把它定义成一个函数？

已知的皇后的位置被传递给 `conflict` 函数(以状态元组的形式)，然后由函数判断下一个的皇后的位置会不会有新的冲突。

```
def conflict(state, nextX):
    nextY = len(state)
    for i in range(nextY):
        if abs(state[i] - nextX) in (0, nextY - i):
            return True
    return False
```

参数 `nextX` 代表下一个皇后的水平位置(`x` 坐标或列)， `nextY` 代表垂直位置(`y` 坐标或行)。这个函数对前面的每个皇后的位置做一个简单的检查，如果下一个皇后和前面的皇后有同样的水平位置，或者是在一条对角线上，就会发生冲突，接着返回 `True` 。如果没有这样的冲突发生，那么返回 `False` ，不太容易理解的是下面的表达式：

```
abs(state[i] - nextX) in (0, nextY - i)
```

如果下一个皇后和正在考虑的前一个皇后的水平距离为0(列相同)或者等于垂直距离(在一条对角线上)就返回 `True` ，否则就返回 `False` 。

9.8.5 基本情况

八皇后问题的实现虽然有点不太好实现，但如果使用生成器就没什么难的了。如果不习惯于使用递归，那么你最好不要自己动手解决这个问题。需要注意的是这个解决方案的效率不是很高，因此如果皇后的数目很多的话，运行起来就会有点慢。

从基本的情况开始：最后一个皇后。你想让它做什么？假设你想找出所有可能的解决方案；这样一来，它能根据其他皇后的为止生成它自己能占据的所有位置(可能没有)。能把这样的情况直接描绘出。

```
def queens(sum, state):
    if len(state) == num - 1:
        for pos in range(num):
            if not conflict(state, pos):
                yield pos
```

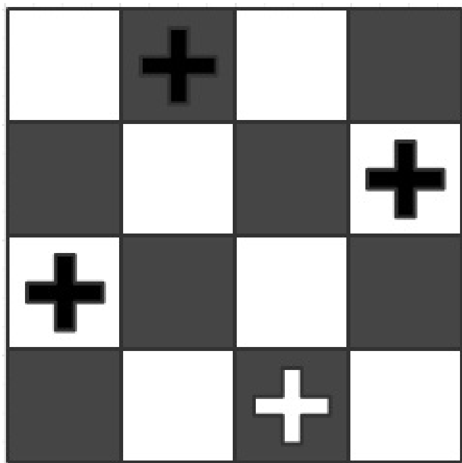
用人类的语言来描述，它的意思是：如果只剩一个皇后没有位置，那么遍历它的所有可能的位置，并且返回没有冲突发生的位置。`num` 参数是皇后的总数。`state` 参数是存放前面皇后的位置信息的元组。假设有4个皇后，前3个分别被放置在1、3、0号位置上，如图9-1所示(不要在意第4行的白色皇后)。

正如在图中看到的，每个皇后占据了一行，并且位置的标号已经到了最大(Python中都是从0开始的)：

```
>>> list(queens(4, (1, 3, 0)))
[2]
```

代码看起来就像一个魔咒。使用 `list` 来让生成器生成列表中的所有值。在这种情况下，只有一个位置是可行的。白色皇后被放置在了如图9-1所示的位置(注意颜色没有特殊含义，不是程序的一部分)。

图9-1 在一个4 x 4的棋盘上放4个皇后



9.8.6 需要递归的情况

现在，让我们看看解决方案中的递归部分。完成基本情况后，递归函数会假定(通过归纳)所有的来自低层(有更高编号的皇后)的结果都是正确的。因此需要做的就是为前面的 `queen` 函数的实现中的 `if` 语句增加 `else` 子句。

那么递归调用会得到什么结果呢？你想得到所有低层皇后的位置，对吗？假设将位置信息作为一个元组返回。在这种情况下，需要修改基本情况也返回一个元组(长度为1)，稍后就会那么做。

这样一来，程序从前面的皇后得到了包含位置信息的元组，并且要为后面的皇后提供当前皇后的每种合法的位置信息。为了让程序继续运行下去，接下来需要做的就是把当前的位置信息添加到元组中并传给后面的皇后。

```
... else:
        for pos in range(num):
            if not conflict(state, pos):
                for result in queens(num, state + (pos, )):
                    yield (pos, ) + result
```

`for pos` 和 `if not conflict` 部分和前面的代码相同，因此可以稍微简化一下代码。添加一些默认的参数：

```
def queens(num=8, state=()):
    for pos in range(num):
        if not conflict(state, pos):
            if len(state) == num - 1:
                yield (pos, )
        else:
            for result in queens(num, state + (pos, )):
                yield (pos, ) + result
```

如果觉得代码很难理解，那么就把代码做的事用自己的语言来叙述，这样能有所帮助。(还记得在 `(pos,)` 中的逗号使其必须被设置为元组而不是简单地加上括号吗？)

生成器 `queens` 能给出所有的解决方案(那就是放置皇后的所有的合法方法)：

```
>>> list(queens(3))
[]
>>> list(queens(4))
[(1, 3, 0, 2), (2, 0, 3, 1)]
>>> for solution in queens(8):
...     print solution
...
(0, 4, 7, 5, 2, 6, 1, 3)
(0, 5, 7, 2, 6, 3, 1, 4)
...
(7, 2, 0, 5, 1, 4, 6, 3)
(7, 3, 0, 2, 5, 1, 6, 4)
```

如果用8个皇后做参数来运行 `queens` ，会看到很多解决方案闪过，来看看有多少种方案：

```
>>> len(list(queens(8)))
92
```

9.8.7 打包

在结束八皇后问题之前，试着将输出处理得更容易理解一点。清理输出总是一个好的习惯，因为这样很容易发现错误。

```
def prettyprint(solution):
    def line(pos, length=len(solution)):
        return ". " * (pos) + "X" + ". " * (length - pos - 1)
        for pos in solution:
            print line(pos)
```

注意 `prettyprint` 中创建了一个小的助手函数。之所以将其放在 `prettyprint` 内，是因为我们假设在外面的任何地方都不会用到它。下面打印出一个令我满意的随机解决方案。可以看到该方案是正确的。

```
>>> import random
>>> prettyprint(random.choice(list(queens(8))))
. . X. . . . .
. . . . . . X. .
. . . . . . . . X
. X. . . . . .
. . . X. . . .
X. . . . . . .
. . . . . . . X.
. . . . X. . .
```

9.9 小结

本章介绍了很多魔法方法，下面来总结一下。

☑ 旧式类和新式类：Python中类的工作方式正在发生变化。目前(3.0版本以前)的Python内有两种类，旧式类已经过时，新式类在2.2版本中被引入，它提供了一些新的特性(比如使用 `super` 函数和 `property` 函数，而旧式类就不能)。为了创建新式类，必须直接或间接子类化 `object` ，或者设置 `__metaclass__` 属性也可以。

☑ 魔法方法：在Python中有一些特殊的方法(名字是以双下划线开始和结束的)。这些方法和函数只有很小的不同，但其中的大部分方法在某些情况下被Python自动调用(比如 `__init__` 在对象被创建后调用)。

☑ 构造方法：这是面向对象的语言共有的，可能要为自己写的每个类实现构造方法。构造方法被命名为 `__init__` 并且在对象被创建后立即自动调用。

☑ 重写：一个类能通过实现方法来重写它的超类中定义的这些方法和属性。如果新方法要调用重写版本的方法，可以从超类(旧式类)直接调用未绑定的版本或使用 `super` 函数(新式类)。

☑ 序列和映射：创建自己的序列或者映射需要实现所有的序列或是映射规则的方法，包括 `__getitem__` 和 `__setitem__` 这样的特殊方法。通过子类化 `list` (或者 `UserList`)和 `dict` (或者 `UserDict`)能节省很多工作。

☑ 迭代器：迭代器是带有 `next` 方法的简单对象。迭代器能在一系列的值上进行迭代。当没有值可供迭代时， `next` 方法就会引发 `StopIteration` 异常。可迭代对象有一个返回迭代器的 `__iter__` 方法，它能像序列那样在 `for` 循环中使用。一般来说，迭代器本身也是可迭代的，即迭代器有返回它自己的 `next` 方法。

☑ 生成器：生成器函数(或者方法)是包含了关键字 `yield` 的函数(或方法)。当被调用时，生成器函数返回一个生成器(一种特殊的迭代器)。可以使用 `send` 、 `throw` 和 `close` 方法让活动生成器和外界交互。

☑ 八皇后的问题：八皇后问题在计算机科学领域内无人不知，使用生成器可以很轻松地解决这个问题。问题描述的是如何在棋盘上放置8个皇后，使其不会互相攻击。

9.9.1 本章的新函数

本章涉及的新函数如表9-1所示。

表**9-1** 本章的新函数

```
iter(obj)                          从一个可迭代对象得到迭代器
property(fget, fset, fdel, doc)    返回一个属性，所有的参数都是可选的
super(class, obj)                  返回一个类的超类的绑定实例
```

注意， `iter` 和 `super` 可能会使用一些其他(未在这里介绍的)参数进行调用。要了解更多的信息，请参见“Standard Python Documentation”(标准Python文档)。

9.9.2 接下来学什么

到目前为止，Python语言的大部分知识都介绍了。那么剩下的那么多章是讲什么的呢？还有很多内容要学，后面的内容很多是关于Python怎么通过各种方法和外部世界联系的。接下来我们还会讨论测试、扩展、打包和一些项目的具体实现，所以请继续努力吧。

第十章 自带电池

> 来源：http://www.cnblogs.com/Marlowes/p/5459376.html
>
> 作者：Marlowes

现在已经介绍了Python语言的大部分基础知识。Python语言的核心非常强大，同时还提供了更多值得一试的工具。Python的标准安装中还包括一组模块，称为标准库(standard library)。之前已经介绍了一些模块(例如 `math` 和 `cmath` ，其中包括了用于计算实数和复数的数学函数)，但是标准库还包含其他模块。本章将向读者展示这些模块的工作方式，讨论如何分析它们，学习它们所提供的功能。本章后面的内容会对标准库进行概括，并且着重介绍一部分有用的模块。

10.1 模块

现在你已经知道如何创建和执行自己的程序(或脚本)了，也学会了怎么用 `import` 从外部模块获取函数并且为自己的程序所用：

```
>>> import math
>>> math.sin(0)
0.0
```

让我们来看看怎样编写自己的模块。

10.1.1 模块是程序

任何Python程序都可以作为模块导入。假设你写了一个代码清单10-1所示的程序，并且将它保存为 `hello.py` 文件(名字很重要)。

```
代码清单10-1 一个简单的模块
# hello.py
print "Hello, world!"
```

程序保存的位置也很重要。下一节中你会了解更多这方面的知识，现在假设将它保存在 `C:\python` (Windows)或者 `~/python` (UNIX/Mac OS X)目录中，接着就可以执行下面的代码，告诉解释器在哪里寻找模块了(以Windows目录为例)：

```
>>> import sys
>>> sys.path.append("c:/python")
```

注：在*UNIX*系统中，不能只是简单地将字符串 `"~/python"` 添加到 `sys.path` 中，必须使用完整的路径*(*例如 `/home/yourusername/python` *)*。如果你希望将这个操作自动完成，可以使用 `sys.path.expanduser("~/python")` 。

我这里所做的知识告诉解释器：除了从默认的目录中寻找之外，还需要从目录 `c:\python` 中寻找模块。完成这个步骤之后，就能导入自己的模块了(存储在 `c:\python\hello.py` 文件中)：

```
>>> import hello
Hello, world!
```

注：在导入模块的时候，你可能会看到有新文件出现——在本例中是 `c:\python\hello.pyc` 。这个以 `.pyc` 为扩展名的文件是*(*平台无关的*)*经过处理*(*编译*)*的，已经转换成*Python*能够更加有效地处理的文件。如果稍后导入同一个模块，*Python*会导入 `.pyc` 文件而不是 `.py` 文件，除非 `.py` 文件已改变，在这种情况下，会生成新的 `.pyc` 文件。删除 `.pyc` 文件不会损害程序*(*只要等效的 `.py` 文件存在即可*)*——必要的时候系统还会创建新的 `.pyc` 文件。

如你所见，在导入模块的时候，其中的代码被执行了。不过，如果再次导入该模块，就什么都不会发生了：

```
>>> import hello
>>>
```

为什么这次没用了？因为导入模块并不意味着在导入时执行某些操作(比如打印文本)。它们主要用于定义，比如变量、函数和类等。此外，因为只需要定义这些东西一次，导入模块多次和导入一次的效果是一样的。

为什么只是一次

这种“只导入一次”(import-only-once)的行为在大多数情况下是一种实质性优化，对于一下情况尤其重要：两个模块互相导入。

在大多数情况下，你可能会编写两个互相访问函数和类的模块以便实现正确的功能。举例来说，假设创建了两个模块—— `clientdb` 和 `billing` ——分别包含了用于客户端数据库和计费系统的代码。客户端数据库可能需要调用计费系统的功能(比如每月自动将账单发送给客户)，而计费系统可能也需要访问客户端数据库的功能，以保证计费准确。

如果每个模块都可以导入数次，那么就出问题了。模块 `clientdb` 会导入 `billing` ，而 `billing` 又导入 `clientdb` ，然后 `clientdb` 又……你应该能想象到这种情况。这个时候导入就成了无限循环。(无限递归，记得吗？)但是，因为在第二次导入模块的时候什么都不会发生，所以循环会终止。

如果坚持重新载入模块，那么可以使用内建的 `reload` 函数。它带有一个参数(需要重新载入的模块)，并且返回重新载入的模块。如果你在程序运行的时候更改了模块并且希望将这些更改反应出来，那么这个功能会比较有用。要重新载入 `hello` 模块(只包含一个 `print` 语句)，可以像下面这样做：

```
>>> hello = reload(hello)
Hello, world!
```

这里假设`hello`已经被导入过(一次)。那么，通过将`reload`函数的返回值赋给`hello`，我们使用重新载入的版本替换了原先的版本。如你所见，问候语已经打印出来了，在此我完成了模块的导入。

如果你已经通过实例化`bar`模块中的`Foo`类创建了一个对象`x`，然后重新载入bar模块，那么不管通过什么方式都无法重新创建引用`bar`的对象`x`，`x`仍然是旧版本`Foo`类的实例(源自旧版本的`bar`)。如果需要x基于重新载入的模块`bar`中的新`Foo`类进行创建，那么你就得重新创建它了。

注意，Python3.0已经去掉了`reload`函数。尽管使用`exec`能够实现同样的功能，但是应该尽可能避免重新载入模块。

10.1.2 模块用于定义

综上所述，模块在第一次导入到程序中时被执行。这看起来有点用——但并不算很有用。真正的用处在于它们(像类一样)可以保持自己的作用域。这就意味着定义的所有类和函数以及赋值后的变量都成为了模块的特性。这看起来挺复杂的，用起来却很简单。

1. 在模块中定义函数

假设我们编写了一个类似代码清单10-2的模块，并且将它存储为hello2.py文件。同时，假设我们将它放置到Python解释器能够找到的地方——可以使用前一节中的`sys.path`方法，也可以用10.1.3节中的常规方法。

注：如果希望模块能够像程序一样被执行(这里的程序是用于执行的，而不是真正作为模块使用的)，可以对Python解释器使用`-m`切换开关来执行程序。如果`progname.py`(注意后缀)文件和其他模块都已被安装(也就是导入了`progname`)，那么运行python `-m progname args`命令就会运行带命令行参数`args`的`progname`程序。

```
代码清单10-2 包含函数的简单模块
# hello2.py
def hello():
    print "Hello, world!"
```

可以像下面这样导入：

```
>>> import hello2
```

模块就会被执行，这意味着`hello`函数在模块的作用域被定义了。因此可以通过以下方式来访问函数：

```
>>> hello2.hello()
Hello, world!
```

我们可以通过同样的方法来使用如何在模块的全局作用域中定义的名称。

我们为什么要这样做呢？为什么不在主程序中定义好一切呢？主要原因是代码重用(code reuse)。如果把代码放在模块中，就可以在多个程序中使用这些代码了。这意味着如果编写了一个非常棒的客户端数据库，并且将它放在叫做`clientdb`的模块中，那么你就可以在计费的时候、发送垃圾邮件的时候（当然我可不希望你这么做）以及任何需要访问客户数据的程序中使用这个模块了。如果没有将这段代码放在单独的模块中，那么就需要在每个程序中重写这些代码了。因此请记住：为了让代码可重用，请将它模块化！(是的，这当然也关乎抽象)

2. 在模块中增加测试代码

模块被用来定义函数、类和其他一些内容，但是有些时候(事实上是经常)，在模块中添加一些检查模块本身是否能正常工作的测试代码是很有用的。举例来说，假如想要确保`hello`函数正常工作，你可能会将`hello2`模块重写为新的模块——代码清单10-3中定义的`hello3`。

```
# hello3.py
def hello():
    print "Hello, world!"

# A test
hello()
```

这看起来是合理的，如果将它作为普通程序运行，会发现它能够正常工作。但如果将它作为模块导入，然后在其他程序中使用hello函数，测试代码就会被执行，就像本章实验开头的第一个`hello`模块一样：

```
>>> import hello3
Hello, world!
>>> hello3.hello.()
Hello, world！
```

这个可不是你想要的。避免这种情况关键在于：“告知”模块本身是作为程序运行还是导入到其他程序。为了实现这一点，需要使用`__name__`变量：

```
>>> __name__
'__main__'
>>> hello3.__name__
'hello3'
```

如你所见，在“主程序”(包括解释器的交互式提示符在内)中，变量`__name__`的值是`'__main__'`。而在导入的模块中，这个值就被设定为模块的名字。因此，为了让模块的测试代码更加好用，可以将其放置在if语句中，如代码清单10-4所示。

```
代码清单10-4 使用条件测试代码的模块 # hello4.py

def hello():
    print "Hello, world!"

def test():
    hello()
if __name__ = '__main__':
    test()
```

如果将它作为程序运行，`hello`函数会被执行。而作为模块导入时，它的行为就会像普通模块一样：

```
>>> import hello4
>>> hello4.hello()
Hello, world!
```

如你所见，我将测试代码放在了`test`函数中，也可以直接将它们放入`if`语句。但是，将测试代码放入独立的`test`函数会更灵活，这样做即使在把模块导入其他程序之后，仍然可以对其进行测试：

```
>>> hello4.test()
Hello, world!
```

注：如果需要编写更完整的测试代码，将其放置在单独的程序中会更好。关于编写测试代码的更多内容，参见第*16*章。

10.1.3 让你的模块可用

前面的例子中，我改变了`sys.path`，其中包含了(字符串组成的)一个目录列表，解释器在该列表中查找模块。然而一般来说，你可能不想这么做。在理想情况下，一开始`sys.path`本身就应该包含正确的目录(包括模块的目录)。有两种方法可以做到这一点：一是将模块放置在合适的位置，另外则是告诉解释器去哪里查找需要的模块。下面几节将探讨这两种方法。

1. 将模块放置在正确位置

将模块放置在正确位置(或者说某个正确位置，因为会有多种可能性)是很容易的。只需要找出Python解释器从哪里查找模块，然后将自己的文件放置在那里即可。

注：如果机器上面的*Python*解释器是由管理员安装的，而你又没有管理员权限，可能无法将模块存储在*Python*使用的目录中。这种情况下，你需要使用另外一个解决方案：告诉解释器去那里查找。

你可能记得，那些(成为搜索路径的)目录的列表可以在`sys`模块中的`path`变量中找到：

```
>>> import sys, pprint
>>> pprint.pprint(sys.path)
['', '/usr/lib/python2.7', '/usr/lib/python2.7/plat-x86_64-linux-gnu', '/usr/lib/pytho
n2.7/lib-tk', '/usr/lib/python2.7/lib-old', '/usr/lib/python2.7/lib-dynload', '/usr/lo
cal/lib/python2.7/dist-packages', '/usr/lib/python2.7/dist-packages', '/usr/lib/python
2.7/dist-packages/PILcompat', '/usr/lib/python2.7/dist-packages/gtk-2.0', '/usr/lib/py
thon2.7/dist-packages/ubuntu-sso-client']
```

注：如果你的数据结构过大，不能在一行打印完，可以使用 *pprint* 模块中的 *pprint* 函数替代普通的 *print* 语句。 *pprint* 是个相当好的打印函数，能够提供更加智能的打印输出。

这是安装在elementary OS上的Python2.7的标准路径，不同的系统会有不同的结果。关键在于每个字符串都提供了一个放置模块的目录，解释器可以从这些目录中找到所需的模块。尽管这些目录都可以使用，但 `site-packages` 目录是最佳的选择，因为它就是用来做这些事情的。查看你自己的 `sys.path` ，找到 `site-packages` 目录，将代码清单10-4的模块存储在其中，要记得改名，比如改成 `another_hello.py` ，然后测试：

```
>>> import another_hello
>>> another_hello.hello()
Hello, world!
```

只要将模块放入类似 `site-packages` 这样的目录中，所有程序就都能将其导入了。

2. 告诉编译器去那里找

“将模块放置在正确的位置”这个解决方案对于以下几种情况可能并不适用：

☑ 不希望将自己的模块填满Python解释器的目录；

☑ 没有在Python解释器目录中存储文件的权限；

☑ 想将模块放在其他地方。

最后一点是“想将模块放在其他地方”，那么就要告诉解释器去哪里找。你之前已经看到了一种方法，就是编辑 `sys.path` ，但这不是通用的方法。标准的实现方法是在 `PYTHONPATH` 环境变量中包含模块所在的目录。

`PYTHONPATH` 环境变量的内容会因为使用的操作系统不同而有所差异(参见下面的“环境变量”)，但基本上来说，它与 `sys.path` 很类似——一个目录列表。

环境变量

环境变量并不是Python解释器的一部分——它们是操作系统的一部分。基本上，它相当于Python变量，不过是在Python解释器外设置的。有关设置的方法，你应该参考操作系统文档，这里只给出一些相关提示。

在UNIX和Mac OS X中，你可以在一些每次登陆都要执行的shell文件内设置环境变量。如果你使用类似bash的shell文件，那么要设置的就是 `.bashrc` ，你可以在主目录中找到它。将下面的命令添加到这个文件中，从而将 `~/python` 加入到 `PYTHONPATH` ：

```
export PYTHON=$PYTHONPATH:~/python
```

注意，多个路径以冒号分隔。其他的shell可能会有不同的语法，所以你应该参考相关的文档。

对于Windows系统，你可以使用控制面板编辑变量(适用于高级版本的Windows，比如Windows XP、2000、NT和Vista，旧版本的，比如Windows 98就不适用了，而需要修改 `autoexec.bat` 文件，下段会讲到)。依次点击开始菜单→设置→控制面板。进入控制面板后，双击“系统”图标。在打开的对话框中选择“高级”选项卡，点击“环境变量”按钮。这时会弹出一个分为上下两栏的对话框：其中一栏是用户变量，另外一栏就是系统变量，需要修改的是用户变量。如果你看到其中已经有 `PYTHONPATH` 项，那么选中它，单击“编辑”按钮进行编辑。如果没有，单击“新建”按钮，然后使用 `PYTHONPATH` 作为“变量名”，输入目录作为“变量值”。注意，多个目录以分号分隔。

如果上面的方法不行，你可以编辑 `autoexec.bat` 文件，该文件可以在C盘的根目录下找到(假设是以标准模式安装的Windows)。用记事本(或者IDLE编辑器)打开它，增加一行设置 `PYTHONPATH` 的内容。如果想要增加目录 `C:\pyrhon` 。可以像下面这样做：

```
set PYTHONPATH=%PYTHONPATH%;C:\python
```

注意，你所使用的IDE可能会有自身的机制，用于设置环境变量和Python路径。

注：你不需要使用 *`PYTHONPATH`* 来更改 *`sys.path`* 。路径配置文件提供了一个有用的捷径，可以让*Python*替你完成这些工作。路径配置文件是以 *`.pth`* 为扩展名的文件，包括应该添加到 *`sys.path`* 中的目录信息。空行和以 *`#`* 开头的行都会被忽略。以 *`import`* 开头的文件会被执行。为了执行路径配置文件，需要将其放置在可以找到的地方。对于*Windows*来说，使用 *`sys.prefix`* 定义的目录名*(*可能类似于 *`C:\Python22`* *)*；在*UNIX*和*Mac OS X*中则使用 *`site-packages`* 目录*(*更多信息可以参见*Python*库参考中 *`site`* 模块的内容，这个模板在*Python*解释器初始化时会自动导入*)*。

3.命名模块

你可能注意到了，包含模块代码的文件的名字要和模块名一样，再加上 `.py` 扩展名。在Windows系统中，你也可以使用 `.pyw` 扩展名。有关文件扩展名含义的更多信息请参见第12章。

10.1.4 包

为了组织好模块，你可以将它们分组为包(package)。包基本上就是另外一个类模块，有趣的地方就是它们能包含其他模块。当模块存储在文件中时(扩展名 `.py`)，包就是模块所在的目录。为了让Python将其作为包对待，它必须包含一个命名为 `__init__.py` 的文件(模块)。如果

将它作为普通模块导入的话，文件的内容就是包的内容。比如有个名为`constants`的包，文件`constants/__init__.py`包括语句`PI=3.14`，那么你可以像下面这么做：

```
import constants
print constants.PI
```

为了将模块放置在包内，直接把模块放在包目录内即可。

比如，如果要建立一个叫做`drawing`的包，其中包括名为`shapes`和`colors`的模块，你就需要创建表10-1中所示的文件和目录(UNIX路径名)。

表10-1 简单的包布局

```
~/python/                              PYTHONPATH中的目录
~/python/drawing/                        包目录(drawing包)
~/python/drawing/__init__.py               包代码(drawing模块)
~/python/drawing/colors.py                colors模块
~/python/drawing/shapes.py                shapes模块
```

对于表10-1中的内容，假定你已经将目录`~/python`放置在`PYTHONPATH`。在Windows系统中，只要用`C:\python`替换`~/python`，并且将正斜线为反斜线即可。

依照这个设置，下面的语句都是合法的：

```
import drawing                    # (1) Imports the drawing package
import drawing.colors             # (2) Imports the colors module
from drawing import shapes        # (3) Imports the shapes module
```

在第1条语句`drawing`中`__init__`模块的内容是可用的，但`drawing`和`colors`模块则不可用。在执行第2条语句之后，`colors`模块可用了，可以通过短名(也就是仅使用`shapes`)来使用。注意，这些语句只是例子，执行顺序并不是必需的。例如，不用像我一样，在导入包的模块前导入包本身，第2条语句可以独立使用，第3条语句也一样。我们还可以在包之间进行嵌套。

10.2 探究模块

在讲述标准库模块前，先教你如何独立地探究模块。这种技能极有价值，因为作为Python程序员，在职业生涯中可能会遇到很多有用的模块，我又不能在这里一一介绍。目前的标准库已经大到可以出本书了(事实上已经有这类书了)，而且它还在增长。每次新的模块发布后，都会添加到标准库，一些模块经常发生一些细微的变化和改进。同时，你还能在网上找到些有用的模块并且可以很快理解(grok)它们，从而让编程轻而易举地称为一种享受。

10.2.1 模块中有什么

探究模块最直接的方式就是在Python解释器中研究它们。当然，要做的第一件事就是导入它。假设你听说有个叫做`copy`的标准模块：

```
>>> import copy
```

没有引发异常，所以它是存在的。但是它能做什么？它又有什么？

1. 使用dir

查看模块包含的内容可以使用`dir`函数，它会将对象的所有特性(以及模块的所有函数、类、变量等)列出。如果想要打印出`dir(copy)`的内容，你会看到一长串的名字(试试看)。一些名字以下划线开始，暗示(约定俗成)它们并不是为在模块外部使用而准备的。所以让我们用列表推导式(如果不记得如何使用了，请参见5.6节)过滤掉它们：

```
>>> [n for n in dir(copy) if not n.startswith("_")]
['Error', 'PyStringMap', 'copy', 'deepcopy', 'dispatch_table', 'error', 'name', 't', '
weakref']
```

这个列表推导式是个包含`dir(copy)`中所有不以下划线开头的名字的列表。它比完整列表要清楚些。(如果喜欢用`tab`实现，那么应该看看库参考中的`readline`和`rlcompleter`模块。它们在探究模块时很有用)

2. `__all__`变量

在上一节中，通过列表推导式所做的事情是推测我可能会在`copy`模块章看到什么。但是我们可以直接从列表本身获得正确答案。在完整的`dir(copy)`列表中，你可能注意到了`__all__`这个名字。这个变量包含一个列表，该列表与我之前通过列表推导式创建的列表很类似——除了这个列表在模块本身中已被默认设置。我们来看看它都包含哪些内容：

```
>>> copy.__all__
['Error', 'copy', 'deepcopy']
```

我的猜测还不算太离谱吧。列表推导式得到的列表只是多出了几个我用不到的名字。但是`__all__`列表从哪来，它为什么会在那儿？第一个问题很容易回答。它是在`copy`模块内部被设置的，像下面这样(从`copy.py`直接复制而来的代码)：

```
__all__ =
["Error", "copy", "deepcopy"]
```

那么它为什么在那呢？它定义了模块的公有接口(public interface)。更准确地说，它告诉解释器：从模块导入所有名字代表什么含义。因此，如果你使用如下代码：

```
from copy import *
```

那么，你就能使用`__all__`变量中的4个函数。要导入`PyStringMap`的话，你就得显示地实现，或者导入`copy`然后使用`copy.PyStringMap`，或者使用`from copy import PyStringMap`。

在编写模块的时候，像设置`__all__`这样的技术是相当有用的。因为模块中可能会有一大堆其他程序不需要或不想要的变量、函数和类，`__all__`会“客气地”将它们过滤了出去。如果没有设定`__all__`，用`import *`语句默认将会导入模块中所有不以下划线开头的全局名称。

10.2.2 用`help`获取帮助

目前为止，你已经通过自己的创造力和Python的多个函数和特殊特性的知识探究了`copy`模块。对于这样的探究工作，交互式解释器是个非常强大的工具，而对该语言的精通程度决定了对模块探究的深度。不过，还有个标准函数能够为你提供日常所需的信息，这个函数叫做`help`。让我们先用`copy`函数试试：

```
>>> help(copy.copy)
Help on function copy in module copy:

copy(x)
    Shallow copy operation on arbitrary Python objects.

    See the module's __doc__ string for more info.
```

这些内容告诉你：`copy`带有一个参数x，并且是“浅复制操作”。但是它还提到了模块的`__doc__`字符串。这是什么呢？你可能记得第六章提到的文档字符串，它就是写在函数开头并且简述函数功能的字符串，这个字符串可以通过函数的`__doc__`特性引用。就像从上面的帮助文本中所理解到的一样，模块也可以有文档字符串(写在模块开头)，类也一样(写在类开头)。

事实上，前面的帮助文本是从`copy`函数的文档字符串中取出的。

```
>>> print copy.copy.__doc__
Shallow copy operation on arbitrary Python objects.

    See the module's __doc__ string for more info.
```

使用`help`与直接检查文档字符串相比，它的好处在于会获得更多信息，比如函数签名(也就是所带的参数)。试着调用`help(copy)`(对模块本身)看看得到什么。它会打印出很多信息，包括`copy`和`deepcopy`之间区别的透彻的讨论(从本质来说，`deepcopy(x)`会将存储在`x`中的值作为属性进行复制，而`copy(x)`只是复制x，将x中的值绑定到副本的属性上)。

10.2.3 文档

模块信息的自然来源当然是文件。我把对文档的讨论推后在这里，是因为自己先检查模块总是快一些。举例来说，你可能会问“`range`的参数是什么”。不用在Python数据或者标准Python文档中寻找有关`range`的描述，而是可以直接查看：

```
>>> print range.__doc__
range(stop) -> list of integers
range(start, stop[, step]) -> list of integers

Return a list containing an arithmetic progression of integers.
range(i, j) returns [i, i+1, i+2, ..., j-1]; start (!) defaults to 0.
When step is given, it specifies the increment (or decrement).
For example, range(4) returns [0, 1, 2, 3].  The end point is omitted!
These are exactly the valid indices for a list of 4 elements.
```

这样就获得了关于 `range` 函数的精确描述，因为Python解释器可能已经运行了(在编程的时候，经常会像这样怀疑函数的功能)，访问这些信息花不了几秒钟。

但是，并非每个模块和函数都有不错的文档字符串(尽管都应该有)，有些时候可能需要十分透彻地描述这些模块和函数是如何工作的。大多数从网上下载的模块都有相关的文档。在我看来，学习Python编程最有用的文档莫过于Python库参考，它对所有标准库中的模块都有描述。如果想要查看Python的知识。十有八九我都会去查阅它。库参考可以在线浏览，并且提供下载，其他一些标准文档(比如Python指南或者Python语言参考)也是如此。所有这些文档都可以在Python网站上找到。

10.2.4 使用源代码

到目前为止，所讨论的探究技术在大多数情况下都已经够用了。但是，对于希望真正理解Python语言的人来说，要了解模块，是不能脱离源代码的。阅读源代码，事实上是学习Python最好的方式，除了自己编写代码外。

真正的阅读不是问题，但是问题在于源代码在哪里。假设我们希望阅读标准模块 `copy` 的源代码，去哪里找呢？一种方案是检查 `sys.pat` h，然后自己找，就像解释器做的一样。另外一种快捷的方法是检查模块的 `__file__` 属性：

```
>>> print copy.__file__
C:\Python27\lib\copy.pyc
```

注：如果文件名以 *.pyc* 结尾，只要查看对应的以 *.py* 结尾的文件即可。

就在那！你可以使用代码编辑器打开 `copy.py` (比如IDLE)，然后查看它是如何工作的。

注：在文本编辑器中打开标准库文件的时候，你也承担着意外修改它的风险。这样做可能会破坏它，所以在关闭文件的时候，你必须确保没有保存任何可能做出的修改。

注意，一些模块并不包含任何可以阅读的Python源代码。它们可能已经融入到解释器内了(比如 `sys` 模块)，或者可能是使用C程序语言写成的(如果模块是使用C语言编写的，你也可以查看它的C源代码)。(请查看第17章以获得更多使用C语言扩展Python的信息)

10.3 标准库：一些最爱

有的读者会觉得本章的标题不知所云。“充电时刻”(batteries included)这个短语最开始由Frank Stajano创造，用于描述Python丰富的标准库。安装Python后，你就“免费”获得了很多有用的模块(充电电池)。因为获得这些模块的更多信息的方式很多(在本章的第一部分已经解释过了)，我不会在这里列出完整的参考资料(因为要占去很大篇幅)，但是我会对一些我最喜欢的标准模块进行说明，从而激发你对模块进行探究的兴趣。你会在“项目章节”(第20章~第29章)碰到更多的标准模块。模块的描述并不完全，但是会强调每个模块比较有趣的特征。

10.3.1 `sys`

`sys` 这个模块让你能够访问与Python解释器联系紧密的变量和函数，其中一些在表10-2中列出。

表10-2 `sys` 模块中一些重要的函数和变量

```
argv                            命令行参数，包括脚本名称
exit([arg])                     退出当前的程序，可选参数为给定的返回值或者错误信息
modules                         映射模块名字到载入模块的字典
path                            查找模块所在目录的目录名列表
platform                        类似sunos5或者win32的平台标识符
stdin                           标准输入流——一个类文件(file-like)对象
stdout                          标准输出流——一个类文件对象
stderr                          标准错误流——一个类文件对象
```

变量 `sys.argv` 包含传递到Python解释器的参数，包括脚本名称。

函数 `sys.exit` 可以退出当前程序(如果在 `try/finally` 块中调用， `finally` 子句的内容仍然会被执行，第八章对此进行了探讨)。你可以提供一个整数作为参数，用来标识程序是否成功运行，这是UNIX的一个惯例。大多数情况下使用该整数的默认值就可以了(也就是0，表示成功)。或者你也可以提供字符串参数，用作错误信息，这对于用户找出程序停止运行的原因会很有用。这样，程序就会在退出的时候提供错误信息和标识程序运行失败的代码。

映射 `sys.modules` 将模块名映射到实际存在的模块上，它只应用于目前导入的模块。

`sys.path` 模块变量在本章前面讨论过，它是一个字符串列表，其中的每个字符串都是一个目录名，在 `import` 语句执行时，解释器就会从这些目录中查找模块。

`sys.platform` 模块变量(它是个字符串)是解释器正在其上运行的“平台”名称。它可能是标识操作系统的名字(比如 `sunos5` 或 `win32`)，也可能标识其他种类的平台，如果运行Jython的话，就是Java的虚拟机(比如 `java1.4.0`)。

`sys.stdin` 、 `sys.stdout` 和 `sys.stderr` 模块变量是类文件流对象。它们表示标准UNIX概念中的标准输入、标准输出和标准错误。简单来说，Python利用 `sys.stdin` 获得输入(比如用于函数 `input` 和 `raw_input` 中的输入)，利用 `sys.stdout` 输出。第十一章会介绍更多有关于文件(以及这三个流)的知识。

举例来说，我们思考一下反序打印参数的问题。当你通过命令行调用Python脚本时，可能会在后面加上一些参数——这就是命令行参数(command-line argument)。这些参数会放置在 `sys.argv` 列表中，脚本的名字为 `sys.argv[0]` 。反序打印这些参数很简单，如代码清单10-5所示。

```
# 代码清单10-5 反序打印命令行参数

import sys

args = sys.argv[1:]
args.reverse()
print " ".join(args)
```

正如你看到的，我对 `sys.argv` 进行了复制。你可以修改原始的列表，但是这样做通常是不安全的，因为程序的其他部分可能也需要包含原始参数的 `sys.argv` 。注意，我跳过了 `sys.argv` 的第一个元素，这是脚本的名字。我使用 `args.reverse()` 方法对列表进行反向排序，但是不能打印出这个操作结果的，这是个返回 `None` 的原地修改操作。下面是另外一种做法：

```
print " ".join(reversed(sys.argv[1:]))
```

最后，为了保证输出得更好，我使用了字符串方法 `join` 。让我们试试看结果如何(我使用的是MS-DOS，在UNIX Shell下它也会工作的同样好)：

```
D:\Workspace\Basic tutorial>python Code10-5.py
this is a test
test a is this
```

10.3.2 `os`

`os` 模块提供了访问多个操作系统服务的功能。 `os` 模块包括的内容很多，表10-3中只是其中一些最有用的函数和变量。另外， `os` 和它的子模块 `os.path` 还包括一些用于检查、构造、删除目录和文件的函数，以及一些处理路径的函数(例如， `os.path.split` 和 `os.path.join` 让你在大部分情况下都可以忽略 `os.pathsep`)。关于它的更多信息，请参见标准库文档。

表10-3 `os` 模块中一些重要函数和变量

```
environ                                        对环境变量进行映射
system(command)                                  在子shell中执行操作系统命令
sep                                           路径中的分隔符
pathsep                                        分隔路径的分隔符
linesep                                       行分隔符("\n", "\r", or "\r\n")
urandom(n)                                     返回n字节的加密强随机数据
```

`os.environ`映射包含本章前面讲述过的环境变量。比如要访问系统变量`PYTHONPATH`，可以使用表达式`os.environ["PYTHONPATH"]`。这个映射也可以用来更改系统环境变量，不过并非所有系统都支持。

`os.system`函数用于运行外部程序。也有一些函数可以执行外部程序。还有`open`，它可以创建与程序连接的类文件。

关于这些函数的更多信息，请参见标准库文档。

注：当前版本的*Python*中，包括*`subprocess`*模块，它包括了*`os.system`*、*`execv`*和*`open`*函数的功能。

`os.sep`模块变量是用于路径名字中的分隔符。UNIX(以及Mac OS X中命令行版本的Python)中的标准分隔符是`"/"`，Windows中的是`"\\"`(即Python针对单个反斜线的语法)，而Mac OS中的是`":"`(有些平台上，`os.altsep`包含可选的路径分隔符，比如Windows中的`"/"`)。

你可以在组织路径的时候使用`os.pathsep`，就像在`PYTHONPATH`中一样。`pathsep`用于分割路径名：UNIX(以及Mac OS X中的命令行版本的Python)使用`":"`，Windows使用`";"`，Mac OS使用`"::"`。

模块变量`os.linesep`用于文本文件的字符串分隔符。UNIX中(以及Mac OS X中命令行版本的Python)为一个换行符(`\n`)，Mac OS中为单个回车符(`\r`)，而在Windows中则是两者的组合(`\r\n`)。

`urandom`函数使用一个依赖于系统的"真"(至少是足够强度加密的)随机数的源。如果正在使用的平台不支持它，你会得到`NotImplementedError`异常。

例如，有关启动网络浏览器的问题。`system`这个命令可以用来执行外部程序，这在可以通过命令行执行程序(或命令)的环境中很有用。例如在UNIX系统中，你可以用它来列出某个目录的内容以及发送Email，等等。同时，它对在图形用户界面中启动程序也很有用，比如网络浏览器。在UNIX中，你可以使用下面的代码(假设`/usr/bin/firefox`路径下有一个浏览器)：

```
os.system("/usr/bin/firefox")
```

以下是Windows版本的调用代码(也同样假设使用浏览器的安装路径)：

```
os.system(r"C:\'Program Files'\'Mozilla Firefox'\firefox.exe")
```

注意，我很仔细地将`Program Files`和`Mozilla Firefox`放入引号中，不然DOS(它负责处理这个命令)就会在空格处停不下来(对于在`PYTHONPATH`中设定的目录来说，这点也同样重要)。同时，注意必须使用反斜线，因为DOS会被正斜线弄糊涂。如果运行程序，你会注意到浏览器会试图打开叫做`Files'\Mozilla...`的网站——也就是在空格后面的命令部分。另一方面，如果试图在IDLE中运行该代码，你会看到DOS窗口出现了，但是没有启动浏览器并没有出现，除非关闭DOS窗口。总之，使用以上代码并不是完美的解决方法。

另外一个可以更好地解决问题的函数是Windows特有的函数—— `os.startfile` ：

```
os.startfile(r"C:\Program Files\Mozilla Firefox\firefox.exe")
```

可以看到， `os.startfile` 接受一般路径，就算包含空格也没问题(也就是不用像在 `os.system` 例子中那样将 `Program Files` 放在引号中)。

注意，在Windows中，由 `os.system` (或者 `os.startfile`)启动了外部程序之后，Python程序仍然会继续运行，而在UNIX中，程序则会中止，等待 `os.system` 命令完成。

更好的解决方案： **`WEBBROWSER`**

在大多数情况下， `os.system` 函数很有用，但是对于启动浏览器这样特定的任务来说，还有更好的解决方案： `webbrowser` 模块。它包括 `open` 函数，它可以自动启动Web浏览器访问给定的URL。例如，如果希望程序使用Web浏览器打开Python的网站(启动新浏览器或者使用已经运行的浏览器)，那么可以使用以下代码：

```
import webbrowser
webbrowser.open("http://www.python.org")
```

10.3.3 `fileinput`

第十一章将会介绍很多读写文件的知识，现在先做个简短的介绍。 `fileinput` 模块让你能够轻松地遍历文本文件的所有行。如果通过以下方式调用脚本(假设在UNIX命令行下)：

```
$ python some_script.py file1.txt file2.txt file3.txt
```

这样就可以以此对 `file1.txt` 到 `file3.txt` 文件中的所有行进行遍历了。你还能对提供给标准输入(`sys.stdin` ，记得吗)的文本进行遍历。比如在UNIX的管道中，使用标准的UNIX命令 `cat` ：

```
$ cat file.txt | python some_script.py
```

如果使用 `fileinput` 模块，在UNIX管道中使用 `cat` 来调用脚本的效果和将文件名作为命令行参数提供给脚本是一样的。 `fileinput` 模块最重要的函数如表10-4所示。

`fileinput.input` 是其中最重要的函数。它会返回能够于 `for` 循环遍历的对象。如果不想使用默认行为(`fileinput` 查找需要循环遍历的文件)，那么可以给函数提供(序列形式的)一个或多个文件名。你还能将 `inplace` 参数设为其真值(inplace=True)以进行原地处理。对于要访问的每一行，需要打印出替代的内容，以返回到当前的输入文件中。在进行原地处理的时候，可选的 `backup` 参数将文件名扩展备份到通过原始文件创建的备份文件中。

表10-4 fileinput模块中重要的函数

```
input(files[, inplace[, backup]])        便于遍历多个输入流中的行
filename()                              返回当前文件的名称
lineno()                                返回当前(累计)的行数
filelineno()                            返回当前文件的行数
isfirstline()                           检查当前行是否是文件的第一行
isstdin()                               检查最后一行是否来自sys.stdin
nextfile()                              关闭当前文件，移动到下一个文件
close()                                 关闭序列
```

`fileinput.filename` 函数返回当前正在处理的文件名(也就是包含了当前正在处理的文本行的文件)。

`fileinput.lineno` 返回当前行的行数。这个数值是累计的，所以在完成一个文件的处理并且开始处理下一个文件的时候，行数并不会重置。而是将上一个文件的最后行数加1作为计数的起始。

`fileinput.filelineno` 函数返回当前处理文件的当前行数。每次处理完一个文件并且开始处理下一个文件时，行数都会重置为1，然后重新开始计数。

`fileinput.isfirstline` 函数在当前行是当前文件的第一行时返回真值，反之返回假值。

`fileinput.isstdin` 函数在当前文件为 `sys.stdin` 时返回真值，否则返回假值。

`fileinput.nextfile` 函数会关闭当前文件，跳到下一个文件，跳过的行并不计。在你知道当前文件已经处理完的情况下，这个函数就比较有用了——比如每个文件都包含经过排序的单词，而你需要查找某个词。如果已经在排序中找到了这个词的位置，那么你就能放心地跳到下一个文件了。

`fileinput.close` 函数关闭整个文件链，结束迭代。

为了演示 `fileinput` 的使用，我们假设已经编写了一个Python脚本，现在想要为其代码进行编号。为了让程序在完成代码行编号之后仍然能够正常运行，我们必须通过在每一行的右侧加上作为注释的行号来完成编号工作。我们可以使用字符串格式化来将代码行和注释排成一行。假设每个程序行最多有45个字符，然后把行号注释加在后面。代码清单10-6展示了使用 `fileinput` 以及 `inplace` 参数来完成这项工作的简单方法：

```
# 代码清单10-6 为Python脚本添加行号

#!/usr/bin/env python
# coding=utf-8

# numberlines.py

import fileinput for line in fileinput.input(inplace=True):
    line = line.rstrip()
    num = fileinput.lineno()
    print "%-40s # %2i" % (line, num)
```

如果你像下面这样在程序本身上运行这个程序：

```
$ python numberline.py numberline.py
```

程序会变成类似于代码清单10-7那样。注意，程序本身已经被更改了，如果这样运行多次，最终会在每一行中添加多个行号。我们可以回忆一下之前的内容：`rstrip`是可以返回字符串副本的字符串方法，右侧的空格都被删除(请参见3.4节，以及附录B中的表B-6)。

```
# 代码清单10-7 为已编号的行进行编号

#!/usr/bin/env python                           # 1
# coding=utf-8                                  # 2
                                                # 3
# numberline.py                                 # 4
                                                # 5
import fileinput                                # 6
                                                # 7
for line in fileinput.input(inplace=True):      # 8
    line = line.rstrip()                        # 9
    num = fileinput.lineno()                    # 10
                                                # 11
    print "%-45s # %2i" % (line, num)           # 12
```

注：要小心使用*`inplace`*参数，它很容易破坏文件。你应该在不使用*`inplace`*设置的情况下仔细测试自己的程序*(*这样只会打印出错误*)*，在确保程序工作正常后再修改文件。

另外一个使用`fileinput`的例子，请参见本章后面的`random`模块部分。

10.3.4 集合、堆和双端队列

在程序设计中，我们会遇到很多有用的数据结构，而Python支持其中一些相对通用的类型，例如字典(或者说散列表)、列表(或者说动态数组)是语言必不可少的一部分。其他一些数据结构尽管不是那么重要，但有些时候也能派上用场。

1. 集合

集合(set)在Python2.3才引入。`Set`类位于`sets`模块中。尽管可以在现在的代码中创建`Set`实例。但是除非想要兼容以前的程序，否则没有什么必要这样做。在Python2.3中，集合通过`set`类型的实例成为了语言的一部分，这意味着不需要导入`sets`模块——直接创建集合即可：

```
>>> set(range(10))
set([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

集合是由序列(或者其他可迭代的对象)构建的。它们主要用于检查成员资格，因此副本是被忽略的：

```
>>> set(["fee", "fie", "foe"])
set(['foe', 'fee', 'fie'])
```

除了检查成员资格外，还可以使用标准的集合操作(可能你是通过数学了解到的)，比如求并集和交集，可以使用方法，也可以使用对整数进行位操作时使用的操作(参见附录B)。比如想要找出两个集合的并集，可以使用其中一个集合的 `union` 方法或者使用按位与(OR)运算符 `"|"` ：

```
>>> a = set([1, 2, 3])
>>> b = set([2, 3, 4])
>>> a.union(b)
set([1, 2, 3, 4])
>>> a | b
set([1, 2, 3, 4])
```

以下列出了一些其他方法和对应的运算符，方法的名称已经清楚地表明了其用途：

```
>>> c = a & b
>>> c.issubset(a)
True
>>> c <= a
True
>>> c.issuperset(a)
False
>>> c >= a
False
>>> a.intersection(b)
set([2, 3])
>>> a & b
set([2, 3])
>>> a.difference(b)
set([1])
>>> a - b
set([1])
>>> a.symmetric_difference(b)
set([1, 4])
>>> a ^ b
set([1, 4])
>>> a.copy()
set([1, 2, 3])
>>> a.copy() is a
False
```

还有一些原地运算符和对应的方法，以及基本方法 `add` 和 `remove` 。关于这方面更多的信息，请参看Python库参考的3.7节 。

注：如果需要一个函数，用于查找并且打印两个集合的并集，可以使用来自 *`set`* 类型的 *`union`* 方法的未绑定版本。这种做法很有用，比如结合 *`reduce`* 来使用：

```
>>> mySets = [] >>> for i in range(10):
...     mySets.append(set(range(i, i + 5)))
...
>>> reduce(set.union, mySets)
set([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13])
```

集合是可变的，所以不能用做字典中的键。另外一个问题就是集合本身只能包含不可变(可散列的)值，所以也就不能包含其他集合。在实际当中，集合的集合是很常用的，所以这个就是个问题了。幸好还有个 `frozenset` 类型，用于代表不可变(可散列)的集合：

```
>>> a = set()
>>> b = set()
>>> a.add(b)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  TypeError: unhashable type: 'set'
>>> a.add(frozenset(b))
>>> a
set([frozenset([])])
```

`frozenset` 构造函数创建给定集合的副本，不管是将集合作为其他集合成员还是字典的键，`frozenset` 都很有用。

2. 堆

另外一个众所周知的数据结构是堆(heap)，它是优先队列的一种。使用优先队列能够以任意顺序增加对象，并且能在任何时间(可能增加对象的同时)找到(也可能是移除)最小的元素，也就是说它比用于列表的 `min` 方法要有效率得多。

事实上，Python中并没有独立的堆类型，只有一个包含一些堆操作函数的模块，这个模块叫做 `heapq` (`q` 是 `queue` 的缩写，即队列)，包括6个函数(参见表10-5)，其中前4个直接和堆操作相关。你必须将列表作为堆对象本身。

表10-5 heapq模块中重要的函数

```
heappush(heap, x)                                  将x入堆
heappop(heap)                                      将堆中最小的元素弹出
heapify(heap)                                      将heap属性强制应用到任意一个列表
heapreplace(heap, x)                               将堆中最小的元素弹出，同时将x入堆
nlargest(n, iter)                                  返回iter中第n大的元素
nsmallset(n, iter)                                 返回iter中第n小的元素
```

`heappush` 函数用于增加堆的项。注意，不能将它用于任何之前讲述的列表中，它只能用于通过各种堆函数建立的列表中。原因是元素的顺序很重要(尽管看起来是随意排列，元素并不是进行严格排序的)。

```
>>> from heapq import *
>>> from random import shuffle
>>> data = range(10)
>>> shuffle(data)
>>> heap = []
>>> for n in data:
...     heappush(heap, n)
...
>>> heap
[0, 2, 1, 6, 5, 3, 4, 9, 7, 8]
>>> heappush(heap, 0.5)
>>> heap
[0, 0.5, 1, 6, 2, 3, 4, 9, 7, 8, 5]
```

元素的顺序并不像看起来那么随意。它们虽然不是严格排序的，但是也有规则的：位于 `i` 位置上的元素总比 `i//2` 位置处的元素大(反过来说就是 `i` 位置处的元素总比 `2*i` 以及 `2*i+1` 位置处的元素小)。这是底层堆算法的基础，而这个特性称为堆属性(heap property)。

`heappop` 函数弹出最小的元素，一般来说都是在索引0处的元素，并且会确保剩余元素中最小的那个占据这个位置(保持刚才提到的堆属性)。一般来说，尽管弹出列表的第一个元素并不是很有效率，但是在这里不是问题，因为 `heappop` 在“幕后”会做一些精巧的移位操作：

```
>>> heappop(heap)
0
>>> heappop(heap)
0.5
>>> heappop(heap)
1
>>> heap
[2, 5, 3, 6, 7, 8, 4, 9]
```

`heapify` 函数使用任意列表作为参数，并且通过尽可能少的移位操作，将其转换为合法的堆(事实上是应用了刚才提到的堆属性)。如果没有用 `heappush` 建立堆，那么在使用 `heappush` 和 `heappop` 前应该使用这个函数。

```
>>> heap = [5, 8, 0, 3, 6, 7, 9, 1, 4, 2]
>>> heapify(heap)
>>> heap
[0, 1, 5, 3, 2, 7, 9, 8, 4, 6]
```

`heapreplace` 函数不像其他函数那么常用。它弹出堆的最小元素，并且将新元素推入。这样做比调用 `heappop` 之后再调用 `heappush` 更高效。

```
>>> heapreplace(heap, 0.5)
0
>>> heap
[0.5, 1, 5, 3, 2, 7, 9, 8, 4, 6]
>>> heapreplace(heap, 10)
0.5
>>> heap
[1, 2, 5, 3, 6, 7, 9, 8, 4, 10]
```

`heapq` 模块中剩下的两个函数 `nlargest(n, iter)` 和 `nsmallest(n, iter)` 分别用来寻找任何可迭代对象 `iter` 中第 `n` 大或第 `n` 小的元素。你可以使用排序(比如使用 `sorted` 函数)和分片来完成这个工作，但是堆算法更快而且更有效第使用内存(还有一个没有提及的有点：更易用)。

3. 双端队列

双端队列(double-ended queue，或称 `deque`)在需要按照元素增加的顺序来移除元素时非常有用，Python2.4增加了 `collection` 模块，它包括 `deque` 类型。

注：*Python2.5*中的 *`collections`* 模块只包括 *`deque`* 类型和 *`defaultdict`* 类型，为不存在的键提供默认值的字典，未来可能会加入二叉树*(B-Tree)*和斐波那契堆*(Fibonacci heap)*。

双端队列通过可迭代对象(比如集合)创建，而且有些非常有用的方法，如下例所示：

```
>>> from collections import deque
>>> q = deque(range(5))
>>> q.append(5)
>>> q.appendleft(6)
>>> q
deque([6, 0, 1, 2, 3, 4, 5])
>>> q.pop()
5
>>> q.popleft()
6
>>> q.rotate(3)
>>> q
deque([2, 3, 4, 0, 1])
>>> q.rotate(-1)
>>> q
deque([3, 4, 0, 1, 2])
```

双端队列好用的原因是它能够有效的在开头(左侧)增加和弹出元素，这是在列表中无法实现的。除此之外，使用双端队列的好处还有：能够有效地旋转(rotate)元素(也就是将它们左移或者右移，使头尾相连)。双端队列对象还有 `extend` 和 `extendleft` 方法， `extend` 和列表的 `extend` 方法差不多， `extendleft` 则类似于 `appendleft` 。注意， `extendleft` 使用的可迭代对象中的元素会反序出现在双端队列中。

10.3.5 `time`

`time` 模块所包括的函数能够实现以下功能：获得当前时间、操作时间和日期、从字符串读取时间以及格式化时间为字符串。日期可以用实数(从“新纪元”的1月1日0点开始计算到现在的秒数，新纪元是一个与平台相关的年份，对UNIX来说是1970年)，或者是包含有9个整数的元组。这些整数的意义如表10-6所示，比如，元组：

```
(2008, 1, 21, 12, 2, 56, 0, 21, 0)
```

表示2008年1月21日12时2分56秒，星期一，并且是当年的第21天(无夏令时)。

表10-6 Python日期元组的字段含义

```
0               年               比如2000,2001等等
1               月               范围1~12
2               日               范围1~31
3               时               范围0~23
4               分               范围0~59
5               秒               范围0~61
6               周               当周一为0时，范围0~6
7               儒历日           范围1~366
8               夏令时           0、1或-1
```

秒的范围是0~61是为了应付闰秒和双闰秒。夏令时的数字是布尔值(真或假)，但是如果使用了 `-1` ， `mktime` (该函数将这样的元组转换为时间戳，它包含从新纪元开始以来的秒数)就会工作正常。 `time` 模块中最重要的函数如表10-7所示。

函数 `time.asctime` 将当前时间格式化为字符串，如下例所示：

```
>>> time.asctime() 'Fri May 13 17:35:56 2016'
```

表10-7 `time` 模块中重要的函数

```
asctime([tuple])                将时间元组转换为字符串
localtime([secs])               将秒数转换为日期元组，以本地时间为准
mktime(tuple)                   将时间元组转换为本地时间
sleep(secs)                     休眠(不做任何事情)secs秒
strptime(string[, format])      将字符串解析为时间元组
time()                          当前时间(新纪元开始后的描述，以UTC为准)
```

如果不需要使用当前时间，还可以提供一个日期元组(比如通过 `localtime` 创建的)。(为了实现更精细的格式化，你可以使用 `strftime` 函数，标准文档对此有相应的介绍)

函数 `time.localtime` 将实数(从新纪元开始计算的秒数)转换为本地时间的日期元组。如果想获得全球统一时间(有关全球统一时间的更多内容，请参见 http://en/wikipedia.org/wiki/Universal_time)，则可以使用 `gtime` 。

函数 `time.mktime` 将日期元组转换为从新纪元开始计算的秒数，它与 `localtime` 的功能相反。

函数 `time.sleep` 让解释器等待给定的秒数。

函数 `time.strptime` 将 `asctime` 格式化过的字符串转换为日期元组(可选的格式化参数所遵循的规则与 `strftime` 的一样，详情请参见标准文档)。

函数 `time.time` 使用自新纪元开始计算的秒数返回当前(全球统一)时间，尽管每个平台的新纪元可能不同，但是你仍然可以通过记录某事件(比如函数调用)发生前后 `time` 的结果来对该事件计时，然后计算差值。有关这些函数的实例，请参见下一节的 `random` 模块部分。

表10-7列出的函数只是从 `time` 模块选出的一部分。该模块的大多数函数所执行的操作与本小节介绍的内容相类似或者相关。如果需要这里没有介绍到的函数，请参见Python库参考的14.2节，以获得更多详细信息。

此外，Python还提供了两个和时间密切相关的模块： `datetime` (支持日期和时间的算法)和 `timeit` (帮助开发人员对代码段的执行时间进行计时)。你可以从Python库参考中找到更多有关它们的信息，第16章也会对 `timeit` 进行简短的介绍。

10.3.6 `random`

`random` 模块包括返回随机数的函数，可以用于模拟或者用于任何产生随机输出的程序。

注：事实上，所产生的数字都是伪随机数，也就是说它们看起来是完全随机的，但实际上，它们以一个可预测的系统作为基础。不过，由于这个系统模块在伪装随机方面十分优秀，所以也就不必对此过多担心了(除非为了实现强加密的目标，因为在这种情况下，这些数字就显

得不够“强”了，无法抵抗某些特定的攻击，但是如果你已经深入到强加密的话，也就不用我来解释这些基础的问题了)。如果需要真的随机数，应该使用*os*模块的 `urandom` 函数。 `random` 模块内的 `SystemRandom` 类也是基于同种功能，可以让数据接近真正的随机性。

这个模块中的一些重要函数如表10-8所示。

表10-8 random模块中的一些重要的函数

```
random()                                                   返回0&lt;n≤1之间的随机实数n
getrandbits(n)                                              以长整型形式返回n个随机位
uniform(a, b)                                               返回随机实数n，其中a≤n&lt;b
randrange([start, ]stop[, step])                               返回range(start, stop, step)中的
随机数
choice(seq)                                                 从序列seq中返回随意元素
shuffle(seq[, random])                                        原地指定序列seq
sample(seq, n)                                              从序列seq中选择n个随机且独立的元素
```

函数 `random.random` 是最基本的随机函数之一，它只是返回0~1的伪随机数 `n` 。除非这就是你想要的，否则你应该使用其他提供了额外功能的的函数。 `random.getrandbits` 以长整型形式返回给定的位数(二进制数)。如果处理的是真正的随机事务(比如加密)，这个函数尤为有用。

为函数 `random.uniform` 提供两个数值参数 `a` 和 `b` ，它会返回在a~b的随机(平均分布的)实数n。所以，比如需要随机数的角度值，可以使用 `uniform(0, 360`)。

调用函数 `range` 可以获得一个范围，而使用与之相同的参数来调用标准函数 `random.randrange` 则能够产生该范围内的随机整数。比如想要获得1~10(包括10)的随机数，可以使用 `randrange(1, 11)` (或者使用 `randrange(10)+1`)，如果想要获得小于20的随机正奇数，可以使用 `randrange(1, 20, 2)` 。

函数 `random.choice` 从给定序列中(均一地)选择随机元素。

函数 `random.shuffle` 将给定(可变)序列的元素进行随机移位，每种排列的可能性都是近似相等的。

函数 `random.sample` 从给定序列中(均一地)选择给定数目的元素，同时确保元素互不相同。

注：从统计学的角度来说，还有些与 `uniform` 类似的函数，它们会根据其他各种不同的分布规则进行抽取，从而返回随机数。这些分布包括贝塔分布、指数分布、高斯分布等等。

下面介绍一些使用 `random` 模块的例子。这些例子将使用一些前文介绍的 `time` 模块中的函数。首先获得代表时间间隔(2008年)限制的实数，这可以通过时间元组的方式来表示日期(使用-1表示一周中的某天，一年中的某天和夏令时，以便让Python自己计算)，并且对这些元组调用 `mktime` ：

```
>>> from random import *
>>> from time import *
>>> date1 = (2008, 1, 1, 0, 0, 0, -1, -1, -1)
>>> time1 = mktime(date1)
>>> date2 = (2009, 1, 1, 0, 0, 0, -1, -1, -1)
>>> time2 = mktime(date2)
```

然后就能在这个范围内均一地生成随机数(不包括上限)：

```
>>> random_time = uniform(time1, time2)
```

然后，可以将数字转换为易读的日期形式：

```
>>> print asctime(localtime(random_time))
Tue Oct 14 04:33:21 2008
```

在接下来的例子中，我们要求用户选择投掷的骰子数以及每个骰子具有的面数。投骰子机制可以由`randrange`和`for`循环实现：

```
#!/usr/bin/env python
# coding=utf-8

from random import randrange

num = input("How many dice? ")
sides = input("How many sides per die? ")
result = 0 for i in range(num):
    result += randrange(sides) + 1

print "The result is", result
```

如果将代码存为脚本文件并且执行，那么会看到下面的交互操作：

```
How many dice? 3
How many sides per die? 6
The result is 11
```

接下来假设有一个新建的文本文件，它的每一行文本都代表一种运势，那么我们就可以使用前面介绍的`fileinput`模块将“运势”都存入列表中，再进行随机选择：

```
# fortunu.py

import fileinput, random

fortunes = list(fileinput.input())
print random.choice(fortunes)
```

在UNIX中，可以对标准字典文件`/usr/dict/words`进行测试，以获得一个随机单词：

```
$ python Code.py /usr/dict/words
Greyson
```

最后一个例子，假设你希望程序能够在每次敲击回车的时候都为自己发一张牌，同时还要确保不会获得相同的牌。首先要创建“一副牌”——字符串列表：

```
>>> values = range(1, 11) + "Jack Queen King".split()
>>> suits = "diamonds clubs hearts spades".split()
>>> deck = ["%s of %s" % (v, s) for v in values for s in suits]
```

现在创建的牌还不太适合进行游戏，让我们来看看现在的牌：

```
>>> from pprint import pprint
>>> pprint(deck[:12])
['1 of diamonds', '1 of clubs', '1 of hearts', '1 of spades', '2 of diamonds', '2 of c
lubs', '2 of hearts', '2 of spades', '3 of diamonds', '3 of clubs', '3 of hearts', '3
of spades']
```

太整齐了，对吧？不过，这个问题很容易解决：

```
>>> from random import shuffle
>>> shuffle(deck)
>>> pprint(deck[:12])
['7 of hearts', 'Queen of hearts', 'Jack of diamonds', '9 of hearts', '2 of diamonds',
 '7 of spades', '10 of diamonds', '8 of diamonds', 'Jack of spades', '4 of spades', '2
 of clubs', 'King of spades']
```

注意，为了节省空间，这里只打印了前12张牌。你可以自己看看整副牌。

最后，为了让Python在每次按回车的时候都给你发一张牌，知道发完为止，那么只需要创建一个小的 `while` 循环即可。假设将建立牌的代码放在程序文件中，那么只需要在程序的结尾处加入下面这行代码：

```
while deck:
    raw_input(deck.pop())
```

注：如果在交互式解释器中尝试上面找到的 *`while`* 循环，那么你会注意到每次按下回车的时候都会打印出一个空字符串。因为 *`raw_input`* 返回了输入的内容*(*什么都没有*)*，并且将其打印出来。在一般的程序中，从 *`raw_input`* 返回的值都会被忽略掉。为了能够在交互环节“忽略”它，只需要把 *`raw_input`* 的值赋给一些你不想再用到的变量即可。同时将这些变量命名为 *`ignore`* 这类名字。

10.3.7 `shelve`

下一章将会介绍如何在文件中存储数据，但如果只需要一个简单的存储方案，那么 `shelve` 模块可以满足你大部分的需要，你所要做的只是为它提供文件名。 `shelve` 中唯一的有趣的函数是 `open` 。在调用它的时候(使用文件名作为参数)，它会返回一个 `shelf` 对象，你10.3.7 `shalve` 可以用它来存储内容。只需要把它当做普通的字典(但是键一定要作为字符串)来操作即可，在完成工作(并且将内容存储到磁盘中)之后，调用它的 `close` 方法。

1. 潜在的陷阱

`shelve.open` 函数返回的对象并不是普通的映射，这一点尤其要注意，如下面的例子所示：

```
>>> import shelve
>>> s = shelve.open("/home/marlowes/workspace/pycharm_Python/Basic_tutorial/test.dat")
>>> s["x"] = ["a", "b", "c"]
>>> s["x"].append("d")
>>> s["x"]
['a', 'b', 'c']
```

`"d"` 去哪了？

很容易解释：当你在 `shelf` 对象中查找元素的时候，这个对象都会根据已经存储的版本进行重新构建，当你将元素赋给某个键的时候，它就被存储了。上述例子中执行的操作如下：

☑ 列表 `["a", "b", "c"]` 存储在键x下。

☑ 获得存储的表示，并且根据它来创建新的列表，而"d"被添加到这个副本中。修改的版本还没有被保存！

☑ 最终，再次获得原始版本——没有 `"d"` 。

为了正确地使用 `shelve` 模块修改存储的对象。必须将临时变量绑定到获得的副本上，并且在它被修改后重新存储这个副本(感谢Luther Blissett指出这个问题)：

```
>>> temp = s["x"]
>>> temp.append("d")
>>> s["x"] = temp
>>> s["x"]
['a', 'b', 'c', 'd']
```

Python2.4之后的版本还有个解决方法：将 `open` 函数的 `writeback` 参数设为true。如果这样做，所有从 `shelf` 读取或者赋值到 `shelf` 的数据结构都会保存在内存(缓存)中，并且只有在关闭 `shelf` 的时候才写回到磁盘中。如果处理的数据不大，并且不想考虑这些问题，那么将 `writeback` 设为 `true` (确保在最后关闭了 `shelf`)的方法还是不错的。

2. 简单的数据库示例

代码清单10-8给出了一个简单的使用shelve模块的数据库应用程序。

```
#!/usr/bin/env python
# coding=utf-8

# database.py

import shelve
def store_person(db):
    """ Query user for data and store it in the shelf object. """
    pid = raw_input("Enter unique ID number: ")
    person = {}
    person["name"] = raw_input("Enter name: ")
    person["age"] = raw_input("Enter age: ")
    person["phone"] = raw_input("Enter phone number: ")
    db[pid] = person
def lookup_person(db):
    """ Query user for ID and desired field, and fetch the correspond data from
    the shelf object. """
    pid = raw_input("Enter ID number: ")
    field = raw_input("What would you like to know? (name, age, phone) ")
    field = field.strip().lower() print field.capitalize() + ":", db[pid][field]
def print_help():
    print "The available commands are:"
    print "store   : Store information about a persoon"
    print "lookup  : Looks up a person from ID number"
    print "quit    : Save changes and exit"
    print "?       : Prints this message"

def enter_command():
    cmd = raw_input("Enter command(? for help): ")
    cmd = cmd.strip().lower()
    return cmd
def main():
    # You may want to change this name
    database = shelve.open("/home/marlowes/workspace/pycharm_Python/Basic_tutorial/dat
abase.dat")
    try:
        while True:
            cmd = enter_command()
            if cmd == "store":
                store_person(database)
            elif cmd == "lookup":
                lookup_person(database)
            elif cmd == "?":
                print_help() elif cmd == "quit": return
    finally:
        database.close()
if __name__ == '__main__':
    main()
```

`Database.py`

代码清单10-8中的程序有一些很有意思的特征。

☑ 将所有内容都放到函数中会让程序更加结构化(可能的改进是将函数组织为类的方法)。

☑ 主程序放在main函数中，只有在 `if __name__ == '__main__'` 条件成立的时候才被调用。这意味着可以在其他程序中将这个程序作为模块导入，然后调用 `main` 函数。

☑ 我在 `main` 函数中打开数据库(`shelf`)，然后将其作为参数传给另外需要它的函数。当然，我也可以使用全局变量，毕竟这个程序很小。不过，在大多数情况下最好避免使用全局变量，除非有充足的理由要使用它。

☑ 在一些值中进行读取之后，对读取的内容调用 `strip` 和 `lower` 函数以生成了一个修改后的版本。这么做的原因在于：如果提供的键与数据库存储的键相匹配，那么它们应该完全一样。如果总是对用户的输入使用 `strip` 和 `lower` 函数，那么就可以让用户随意输入大小写字母和添加空格了。同时需要注意的是：在打印字段名称的时候，我使用了 `capitalize` 函数。

☑ 我使用 `try/finally` 确保数据库能够正确关闭。我们永远不知道什么时候会出错(同时程序会抛出异常)。如果程序在没有正确关闭数据库的情况下终止，那么，数据库文件就有可能被损坏了，这样的数据文件是毫无用处的。使用 `try/finally` 就可以避免这种情况了。

接下来，我们测试一下这个数据库。下面是一个简单的交互过程：

```
Enter command(? for help): ?
The available commands are:
store   : Store information about a persoon
lookup  : Looks up a person from ID number
quit    : Save changes and exit
?       : Prints this message
Enter command(? for help): store
Enter unique ID number: 001 Enter name: Greyson
Enter age: 19 Enter phone number: 001-160309 Enter command(? for help): lookup
Enter ID number: 001 What would you like to know? (name, age, phone) phone
Phone: 001-160309 Enter command(? for help): quit
```

交互的过程并不是十分有趣，使用普通的字典也能获得和 `shelf` 对象一样的效果。但是，我们现在退出程序，然后再重新启动它，看看发生了什么？也许第二天才重新启动它：

```
Enter command(? for help): lookup
Enter ID number: 001 What would you like to know? (name, age, phone) name
Name: Greyson
Enter command(? for help): quit
```

我们可以看到，程序读出了第一次创建的文件，而Greyson的资料还在！

你可以随意试验这个程序，看看是否还能扩展它的功能并且提高用户友好度。你是不是想创建一个供自己使用的版本？创建一个唱片集的数据库怎样？或者创建一个数据库，帮助自己记录借书朋友的名单(我想我会用这个版本)。

10.3.8 `re`

有些人面临一个问题时回想："我知道，可以使用正则表达式来解决这个问题。"于是现在他们就有两个问题了。 ——Jamie Zawinski（Lisp黑客，Netscape早期开发者。关于他的更详细编程生涯，可见人民邮电出版社出版的《编程人生》一书）

`re` 模块包含对正则表达式(regular expression)的支持。如果你之前听说过正则表达式，那么你可能知道它有多强大了，如果没有，请做好心里准备吧，它一定会令你很惊讶。

但是应该注意，在学习正则表达式之初会有点困难(好吧，其实是很难)。学习它们的关键是一次只学习一点——(在文档中)查找满足特定任务需要的那部分内容，预先将它们全部记住是没必要的。本章将会对 `re` 模块主要特征和正则表达式进行介绍，以便让你上手。

注：除了标准文档外，*Andrew Kuchling*的*"Regular Expression HOWTO"*（正则表达式*HOWTO*）也是学习在*Python*中使用正则表达式的有用资源。

1.什么是正则表达式

正则表达式是可以匹配文本片段的模式。最简单的正则表达式就是普通字符串，可以匹配其自身。换句话说，正则表达式"python"可以匹配字符串"python"。你可以用这种匹配行为搜索文本中的模式，并且用计算后的值替换特定模式，或者将文本进行分段。

○ 通配符

正则表达式可以可以匹配多于一个的字符串，你可以使用一些特殊字符串创建这类模式。比如点号(`.`)可以匹配任何字符(除了换行符)，所以正则表达式 `".ython"` 可以匹配字符串 `"python"` 和 `"jython"` 。它还能匹配 `"qython"` 、 `"+ython"` 或者 `" ython"` (第一个字母是空格)，但是不会匹配 `"cpython"` 或者 `"ython"` 这样的字符，因为点号只能匹配一个字母，而不是两个或者零个。

因为它可以匹配“任何字符串”(除换行符外的任何单个字符)，点号就称为通配符(wildcard)。

○ 对特殊字符进行转义

你需要知道：在正则表达式中如果将特殊字符作为普通字符使用会遇到问题，这很重要。比如，假设需要匹配字符串 `"python.org"` ，直接调用 `"python.org"` 可以么？这么做是可以的，但是这样也会匹配 `"pythonzorg"` ，这可不是所期望的结果(点号可以匹配除换行符外的任何字符，还记得吧)。为了让特殊字符表现得像普通字符一样，需要对它进行转义(escape)，就像我在第1章中对引号进行转义所做的一样——可以在它前面加上反斜线。因此，在本例中可以使用 `"python\\.org"` ，这样就只会匹配 `"python.org"` 了。

注：为了获得 `re` 模块所需的单个反斜线，我们要在字符串中使用两个反斜线——为了通过解释器进行转义。这样就需要两个级别的转义了：*(1)*通过解释器转义；*(2)*通过*re*模块转义*(*事实上，有些情况下可以使用单个反斜线，让解释器自动进行转义，但是别依赖这种功能*)*。如果厌烦了使用双斜线，那么可以使用原始字符串，比如 `r"python\.org"` 。

○ 字符集

匹配任意字符可能很有用，但有些时候你需要更多的控制权。你可以使用中括号括住字符串来创建字符集(character set)。字符集可以匹配它所包括的任意字符，所以 `"[pj]ython"` 能够匹配 `"python"` 和 `"jython"` ，而非其他内容。你可以使用范围，比如 `"[a-z]"` 能够(按字母顺序)匹配 `a` 到 `z` 的任意一个字符，还可以通过一个接一个的方式将范围联合起来使用，比如 `"[a-zA-Z0-9]"` 能够匹配任意大小写字母和数字(注意字符集只能匹配一个这样的字符)。

为了反转字符集，可以在开头使用^字符，比如 `"[^abc]"` 可以匹配任何除了 `a` 、 `b` 和 `c` 之外的字符。

字符集中的特殊字符

一般来说，如果希望点号、星号和问号等特殊字符在模式中用作文本字符而不是正则表达式运算符，那么需要用反斜线进行转义。在字符集中，对这些字符进行转义通常是没必要的(尽管是完全合法的)。不过，你应该记住下面的规则：

☑ 如果脱字符(`^`)出现在字符集的开头，那么你需要对其进行转义了，除非希望将它用做否定运算符(换句话说，不要将它放在开头，除非你希望那样用)；

☑ 同样，右中括号(`]`)和横线(`-`)应该放在字符集的开头或者用反斜线转义(事实上，如果需要的话，横线也能放在末尾)。

○ 选择符和子模式

在字符串的每个字符都有各不相同的情况下，字符集是很好用的，但如果只想匹配字符串 `"python"` 和 `"perl"` 呢？你就不能使用字符集或者通配符来指定某个特定的模式了。取而代之的是用于选择项的特殊字符：管道符号(|)。因此，所需的模式可以写成 `"python|perl"` 。

但是，有些时候不需要对整个模式使用选择运算符，只是模式的一部分。这时可以使用圆括号括起需要的部分，或称子模式(subparttern)。前例可以写成 `"p(ython|erl)"` 。(注意，术语子模式也是适用于单个字符)

○ 可选项和可重复子模式

在子模式后面加上问号，它就变成了可选项。它可能出现在匹配字符串中，但并非必需的。例如，下面这个(稍微有点难懂)模式：

```
r"(http://)?(www\.)?python\.org"
```

只能匹配下列字符串(而不会匹配其他的)：

```
"http://www.python.org"
"http://python.org"
"www.python.org"
"python.org"
```

对于上述例子，下面这些内容是值得注意的：

☑ 对点号进行了转义，防止它被作为通配符使用；

☑ 使用原始字符串减少所需反斜线的数量；

☑ 每个可选子模式都用圆括号括起；

☑ 可选子模式出现与否均可，而且互相独立。

问号表示子模式可以出现一次或根本不出现，下面这些运算符允许子模式重复多次：

☑ `(pattern)*` ：允许模式重复0次或多次；

☑ `(pattern)+` ：允许模式重复1次或多次；

☑ `(patten){m,n}` ：允许模式重复m~n次。

例如， `r"w*\.python\.org"` 会匹配 `"www.python.org"` ，也会匹配 `".python.org"` 、 `"ww.python.org"` 和 `"wwwwww.python.org"` 。类似地， `r"w+\.python\.org"` 匹配 `"w.python.org"` 但不匹配 `".python.org"` ，而 `r"w{3,4}\.python\.org"` 只匹配 `"www.python.org"` 和 `"wwww.python.org"` 。

注：这里使用术语匹配*(match)*表示模式匹配整个字符串。而接下来要说到的*match*函数*(*参见表*10-9)*只要求模式匹配字符串的开始。

○ 字符串的开始和结尾

目前为止，所出现的模式匹配都是针对整个字符串的，但是也能寻找匹配模式的子字符串，比如字符串 `"www.python.org"` 中的子字符串 `"www"` 能够匹配模式 `"w+"` 。在寻找这样的子字符串时，确定子字符串位于整个字符串的开始还是结尾是很有用的。比如，只想在字符串的开头而不是其他位置匹配 `"ht+p"` ，那么就可以使用脱字符(`^`)标记开始： `"^ht+p"` 会匹配 `"http://python.org"` (以及 `"htttp://python.org"`)，但是不匹配 `"www.python.org"` 。类似的，字符串结尾用美元符号(`$`)标识。

注：有关正则表达式运算符的完整列表，请参见*Python*类参考的*4.2.1*节的内容。

2. `re` 模块的内容

如果不知道如何应用，只知道如何书写正则表达式还是不够的。 `re` 模块包含一些有用的操作正则表达式的函数。其中最重要的一些函数如表10-9所示。

表10-9 `re` 模块中一些重要的函数

```
compile(pattern[, flags])                          根据包含正则表达式的字符串创建模式对象
search(pattern， string[, flags])                   在字符串中寻找模式
match(pattern, string[, flags])                     在字符串的开始处匹配模式
split(pattern string[, maxsplit=0])                  根据模式的匹配项来分割字符串
findall(pattern, string)                           列出字符串中模式的所有匹配项
sub(pat, repl, string[, count=0])                    将字符串中所有pat的匹配项用repl替换
escape(string)                                   将字符串中所有特性正则表达式字符转义
```

函数 `re.compile` 将正则表达式(以字符串书写的)转换成模式对象，可以实现更有效率的匹配。如果在调用 `search` 或者 `match` 函数的时候使用字符串表示的正则表达式，它们也会在内部将字符串转换为正则表达式对象。使用 `compile` 完成一次转换之后，在每次使用模式的时候就不用进行转换。模式对象本身也没有查找/匹配的函数，就像方法一样，所以 `re.search(pat, string)` (`pat` 是用字符串表示的正则表达式)等价于 `pat.search(string)` (`pat` 是用 `compile` 创建的模式对象)。经过 `compile` 转换的正则表达式对象也能用于普通的 `re` 函数。

函数 `re.search` 会在给定字符串中寻找第一个匹配给定正则表达式的子字符串。一旦找到子字符串，函数就会返回 `MatchObject` (值为 `True`)，否则返回 `None` (值为 `False`)。因为返回值的性质，所以该函数可以用在条件语句中，如下例所示：

```
if re.search(pat, string): print "Found it!"
```

同时，如果需要更多有关匹配的子字符串的信息，那么可以检查返回的 `MatchObject` 对象(有关 `MatchObject` 更多的内容，请参见下一节)。

函数 `re.match` 会在给定字符串的开头匹配正则表达式。因此， `match("p", "python")` 返回眞(即匹配对象 `MatchObject`)，而 `re.match("p", "www.python.org")` 则返回假(`None`)。

注：如果模式与字符串的开始部分相匹配，那么 *`match`* 函数会给出匹配的结果，而模式并不需要匹配整个字符串。如果要求模式匹配整个字符串，那么可以在模式的结尾加上美元符号。美元符号会对字符串的末尾进行匹配，从而“顺延”了整个匹配。

函数 `re.split` 会根据模式的匹配项来分割字符串。它类似于字符串方法 `split` ，不过是用完整的正则表达式替代了固定的分隔符字符串。比如字符串方法 `split` 允许用字符串 `","` 的匹配项来分割字符串，而 `re.split` 则允许用任意长度的逗号和空格序列来分割字符串：

```
>>> import re
>>> some_text = "alpha, beta,,,,gamma delta"
>>> re.split("[, ]+", some_text)
['alpha', 'beta', 'gamma', 'delta']
```

注：如果模式包含小括号，那么括起来的字符组合会散布在分割后的子字符串之间。例如， *`re.split("o(o)", "foobar")`* 回生成 *`["f", "o", "bar"]`* 。

从上述例子可以看到，返回值是子字符串的列表。 `maxsplit` 参数表示字符串最多可以分割的次数：

```
>>> re.split("[, ]+", some_text, maxsplit=2)
['alpha', 'beta', 'gamma delta']
>>> re.split("[, ]+", some_text, maxsplit=1)
['alpha', 'beta,,,,gamma delta']
```

函数 `re.findall` 以列表形式返回给定模式的所有匹配项。比如，要在字符串中查找所有的单词，可以像下面这么做：

```
>>> pat = "[a-zA-Z]+"
>>> text = '"Hm... Err -- are you sure?" he said, sounding insecure.'
>>> re.findall(pat, text)
['Hm', 'Err', 'are', 'you', 'sure', 'he', 'said', 'sounding', 'insecure']
```

或者查找标点符号：

```
>>> pat = r'[.?\-",]+'
>>> re.findall(pat, text)
['"', '...', '--', '?"', ',', '.']
```

注意，横线(`-`)被转义了，所以Python不会将其解释为字符范围的一部分(比如a~z)。

函数`re.sub`的作用在于：使用给定的替换内容将匹配模式的子字符串(最左端并且非重叠的子字符串)替换掉。请思考下面的例子：

```
>>> pat = '{name}'
>>> text = 'Dear {name}...'
>>> re.sub(pat, "Mr. Greyson", text) 'Dear Mr. Greyson...'
```

请参见本章后面“作为替换的组号和函数”部分，该部分会向你介绍如何更有效地使用这个函数。

`re.escape`是一个很实用的函数，它可以对字符串中所有可能被解释为正则运算符的字符进行转义的应用函数。如果字符串很长且包含很多特殊字符，而你又不想输入一大堆反斜线，或者字符串来自于用户(比如通过`raw_input`函数获取的输入内容)，且要用作正则表达式的一部分的时候，可以使用这个函数。下面的例子向你演示了该函数是如何工作的：

```
>>> re.escape("www.python.org")
'www\\.python\\.org'
>>> re.escape("But where is the ambiguity?")
'But\\ where\\ is\\ the\\ ambiguity\\?'
```

注：你可能会注意到，表*10-9*中有些函数包含了一个名为*`flags`*的可选参数。这个参数用于改变解释正则表达式的方法。有关它的更多信息，请参见*Python库参考的4.2节*。这个标志在*4.2.3*节中有介绍。

3.匹配对象和组

对于`re`模块中那些能够对字符串进行模式匹配的函数而言，当能找到匹配项的时候，它们都会返回`MatchObject`对象。这些对象包括匹配模式的子字符串的信息。它们还包含了那个模式匹配了子字符串哪部分的信息——这些“部分”叫做组(group)。

简而言之，组就是放置在圆括号内的子模式。组的序号取决于它左侧的括号数。组0就是整个模式，所以在下面的模式中：

```
"There (was a (wee) (cooper)) who (lived in Fyfe)"
```

包含下面这些组：

```
0 There was a wee cooper who lived in Fyfe
1 was a wee cooper
2 wee
3 cooper
4 lived in Fyfe
```

一般来说，如果组中包含诸如通配符或者重复运算符之类的特殊字符，那么你可能会对是什么与给定组实现了匹配感兴趣，比如在下面的模式中：

```
r"www\.(.+)\.com$"
```

组0包含整个字符串，而组1则包含位于 `"www."` 和 `".com"` 之间的所有内容。像这样创建模式的话，就可以取出字符串中感兴趣的部分了。

`re` 匹配对象的一些重要方法如表10-10所示。

表10-10 re匹配对象的重要方法

```
group([group1, ...])      获取给定子模式(组)的匹配项
start([group])           返回给定组的匹配项的开始位置
end([group])             返回给定组的匹配项的结束位置(和分片不一样，不包括组的结束位置)
span([group])            返回一个组的开始和结束位置
```

`group` 方法返回模式中与给定组匹配的(子)字符串。如果没有给出组号，默认为组0。如果给定一个组号(或者只用默认的0)，会返回单个字符串。否则会将对应给定组数的字符串作为元组返回。

注：除了整体匹配外*(组0)*，我们只能使用*99*个组，范围*1~99*。

`start` 方法返回给定组匹配项的开始索引(默认为0，即整个模式)。

方法 `end` 类似于 `start` ，但是返回结果是结束索引加1。

方法 `span` 以元组 `(start,end)` 的形式返回给定组的开始和结束位置的索引(默认为0，即整个模式)。

请思考以下例子：

```
>>> m = re.match(r"www\.(.*)\..{3}", "www.python.org")
>>> m.group(1)
'python'
>>> m.start(1)
4
>>> m.end(1)
10
>>> m.span(1)
(4, 10)
```

4. 作为替换的组号和函数

在使用`re.sub`的第一个例子中，我只是把一个字符串用其他的内容替换掉了。我用`replace`这个字符串方法(3.4节对此进行了介绍)能轻松达到同样的效果。当然，正则表达式很有用，因为它们允许以更灵活的方式搜索，同时它们也允许进行功能更强大的替换。

见证`re.sub`强大功能的最简单方式就是在替换字符串中使用组号。在替换内容中以`"\\n"`形式出现的任何转义序列都会被模式中与组n匹配的字符串替换掉。例如，假设要把`"*something*"`用`"<em>something</em>"`替换掉，前者是在普通文本文档(比如Emaill)中进行强调的常见方法，而后者则是相应的HTML代码(用于网页)。我们首先建立正则表达式：

```
>>> emphasis_pattern = r"\*([^\*]+)\*"
```

注意，正则表达式很容易变得难以理解，所以为了让其他人(包括自己在内)在以后能够读懂代码，使用有意义的变量名(或者加上一两句注释)是很重要的：

注：让正则表达式变得更加易读的方式是在`re`函数中使用`VERBOSE`标志。它允许在模式中添加空白(空白字符、`tab`、换行符，等等)，`re`则会忽略它们，除非将其放在字符类或者用反斜线转义。也可以在冗长的正则式中添加注释。下面的模式对象等价于刚才写的模式，但是使用了`VERBOSE`标志：

```
>>> emphasis_pattern = re.compile(r'''
...     \*         # Beginning emphasis tag -- an asterisk
...     (          # Begin group for capturing phrase
...     [^\*]+     # Capture anything except asterisks
...     )          # End group
...     \*         # Ending emphasis tag
... ''', re.VERBOSE)
```

现在模式已经搞定，接下来就可以使用re.sub进行替换了：

```
>>> re.sub(emphasis_pattern, r"<em>\1</em>", "Hello, *world*!")
'Hello, <em>world</em>!'
```

从上述例子可以看到，普通文本已经成功地转换为HTML。

将函数作为替换内容可以让替换功能变得更加强大。`MatchObject`将作为函数的唯一参数，返回的字符串将会用做替换内容。换句话说，可以对匹配的子字符串做任何事，并且可以细化处理过程，以生成替换内容。你可能会问，这个功能用在什么地方呢？开始使用正则表达式以后，你肯定会发现这个功能的无数应用。本章后面的"模板系统示例"部分会向你介绍它的一个应用。

贪婪和非贪婪模式

重复运算符默认是贪婪(greedy)的，这意味着它会进行尽可能多的匹配。比如，假设我重写了刚才用到的程序，以使用下面的模式：

```
>>> emphasis_pattern = r"\*(.+)\*"
```

它会匹配星号加上一个或多个字符，再加上一个星号的字符串。听起来很完美吧？但实际上不是：

```
>>> re.sub(emphasis_pattern, r"<em>\1</em>", "*This* is *it*!")
'<em>This* is *it</em>!'
```

模式匹配了从开始星号到结束星号之间的所有内容——包括中间的两个星号！也就意味着它是贪婪的：将尽可能多的东西都据为己有。

在本例中，你当然不希望出现这种贪婪行为。当你知道某个特定字母不合法的时候，前面的解决方案(使用字符集匹配任何不是星号的内容)才是可行的。但是假设另外一种情况：如果使用 `"**something**"` 表示强调呢？现在在所强调的部分包括单个星号已经不是问题了，但是如何避免过于贪婪？

事实上非常简单，只要使用重复运算符的非贪婪版本即可。所有的重复运算符都可以通过在其后面加上一个问号变成非贪婪版本：

```
>>> emphasis_pattern = r"\*\*(.+?)\*\*"
>>> re.sub(emphasis_pattern, r"<em>\1</em>", "**This** is **it**!")
'<em>This</em> is <em>it</em>!'
```

这里用 `+?` 运算符代替了 `+` ，意味着模式也会像之前那样队一个或者多个通配符进行匹配，但是它会进行尽可能少的匹配，因为它是非贪婪的。它仅会在到达 `"\*\*"` 的下一个匹配项之前匹配最少的内容——也就是在模式的结尾进行匹配。我们可以看到，代码工作得很好。

5. 找出Email的发信人

有没有尝试过将Email存为文本文件？如果有的话，你会看到文件的头部包含了一大堆与邮件内容无关的信息，如代码清单10-9所示。

```
#代码清单10-9 一组(虚构的)Email头部信息
 From foo@bar.baz Thu Dec 20 01:22:50 2008 Return-Path: <foo@bar.baz> Received: from x
yzzy42.bar.com (xyzzy.bar.baz [123.456.789.42])
        by frozz.bozz.floop (8.9.3/8.9.3) with ESMTP id BAA25436 for <maguns@bozz.floo
p>: Thu 20 Dec 2004 01:22:50 +0100 (MET)
Received: from [43.253.124.23] by bar.baz
          [InterMail vM.4.01.03.27 201-229-121-20010626] with ESMTP
          id <20041220002242.ADASD123.bar.baz@[43.253.124.23]>:
          Thu, 20 Dec 2004 00:22:42 +0000 User-Agent: Microsot-Outlook-Express-Macinto
sh-Edition/5.02.2022 Date: Wed, 19 Dec 2008 17:22:42 -0700 Subject: Re: Spam
From: Foo Fie <foo@bar.baz> To: Magnus Lie Hetland <magnus@bozz.floop> CC: <Mr.Gumby@b
ar.baz> Message-ID: <B8467D62.84F%foo@baz.com> In-Reply-To: <20041219013308.A2655@bozz
.floop> Mime-version: 1.0 Content-type: text/plain: charset="US-ASCII" Content-transfe
r-encoding: 7bit
Status: RO
Content-Length: 55 Lines: 6 So long, and thanks for all the spam!

Yours.

Foo Fie
```

我们试着找出这封Email是谁发的。如果直接看文本，你肯定可以指出本例中的发信人(特别是查看邮件结尾签名的话，那就更直接了)。但是能找出通用的模式吗？怎么能把发信人的名字取出而不带着Email地址呢？或者如何将头部信息中包含的Email地址列示出来呢？我们先处理第一个任务。

包含发信人的文本行以字符串`"From:"`作为开始，以放置在尖括号(`<`和`>`)中的Email地址作为结束。我们需要的文本就夹在中间。如果使用`fileinput`模块，那么这个需求就很容易实现了。代码清单10-10给出了解决这个问题的程序。

注：这个问题也可以不使用正则表达式解决，可以使用*`email`*模块。

```
# 代码清单10-10 寻找Email发信人的程序

# RegularExpression.py
import fileinput import re

pat = re.compile(r"From: (.*) <.*?>$")
for line in fileinput.input():
    m = pat.match(line) if m: print m.group(1)
```

可以像下面这样运行程序(假设邮件内容存储在文本文件`message.eml`中)：

```
$ python RegularExpression.py message.eml
Foo Fie
```

对于这个程序，应该注意以下几点：

☑ 我用`compile`函数处理了正则表达式，让处理过程更有效率；

☑ 我将需要取出的子模式放在圆括号中作为组；

☑ 我使用非贪婪模式对邮件地址进行匹配，那么只有最后一对尖括号符合要求(当名字包含了尖括号的情况下)；

☑ 我使用了美元符号表明我要匹配正行；

☑ 我使用if语句确保在我试图从特定组中取出匹配内容之前，的确进行了匹配。

为了列出头部信息中所有的Email地址，需要建立只匹配Email地址的正则表达式。然后可以使用`findall`方法寻找每行出现的匹配项。为了避免重复，可以将地址保存在集合中(本章前面介绍过)。最后，取出所有的键，排序，并且打印出来：

```
import re import fileinput

pat = re.compile(r"[a-z\-\.]+@[a-z\-\.]+", re.IGNORECASE)
addresses = set()
for line in fileinput.input():
    for address in pat.findall(line):
        addresses.add(address)
    for address in sorted(addresses):
        print address
```

运行程序的时候会输出如下结果(以代码清单10-9的邮件信息作为输入)：

```
Mr.Gumby@bar.baz
foo@bar.baz
foo@baz.com
magnus@bozz.floop
```

注：在这里，我并没有严格照着问题规范去做。问题的要求是在头部找出*Email*地址，但是这个程序找出了整个文件中的地址。为了避免这种情况，如果遇到空行就可以调用 `fileinput.close()` ，因为头部不包含空行，遇到空行就证明工作完成了。此外，你还可以使用 `fileinput.nextfile()` 开始处理下一个文件——如果文件多于一个的话。

6. 模板系统示例

模板是一种通过放入具体值从而得到某种已完成文本的文件。比如，你可能会有只需要插入收件人姓名的邮件模板。Python有一种高级的模板机制：字符串格式化。但是使用正则表达式可以让系统更加高级。假设需要把所有 `"[somethings]"` (字段)的匹配项替换为通过Python表达式计算出来的 `something` 结果，所以下面的字符串：

```
"The sum of 7 and 9 is [7 + 9]."
```

应该被翻译为如下形式：

```
"The sum of 7 and 9 is 16."
```

同时，还可以在字段内进行赋值，所以下面的字符串：

```
"[name='Mr. Gumby']Hello, [name]"
```

应该被翻译为如下形式：

```
"Hello, Mr. Gumby"
```

看起来像是复杂的工作，但是我们再看一下可用的工具。

☑ 可以使用正则表达式匹配字段，提取内容。

☑ 可以用 `eval` 计算字符值，提供包含作用域的字典。可以在 `try/except` 语句内进行这项工作。如果引发了 `SyntaxError` 异常，可能是某些语句出现了问题(比如赋值)，应该使用 `exec` 来代替。

☑ 可以用 `exce` 执行字符串(和其他语句)的赋值操作，在字典中保存模板的作用域。

☑ 可以使用 `re.sub` 将求值的结果替换为处理后的字符串。

这样看来，这项工作又不再让人寸步难行了，对吧？

注：如果某项任务令人望而却步，将其分解为小一些的部分总是有用的。同时，要对解决问题所使用的工具进行评估。

代码清单10-11是一个简单的实现。

```
#!/usr/bin/env python # coding=utf-8

# templates.py

import re import fileinput
# Matching in brackets in the field.
filed_pat = re.compile(r"\[(.+?)\]")
# We will be variable collected here
scope = {}
# Used in the re.sub.
def replacement(math):
    code = math.group(1)
    try:
        # If the field can be evaluated, then return it.
        return str(eval(code, scope))
    except SyntaxError:
        # Otherwise the same scope of assignment statements.
        exec code in scope
        # Return an empty string.
        return ""

# All text in the from of a string.
# There are other ways, see chapter 11.
lines = []
for line in fileinput.input():
    lines.append(line)

text = "".join(lines)
# Replace all field pattern match.
print filed_pat.sub(replacement, text)
```

Templates.py

简单来说，程序做了下面的事情。

☑ 定义了用于匹配字段的模式。

☑ 创建充当模板作用域的字典。

☑ 定义具有下列功能的替换函数。

  * 将组1从匹配中取出，放入`code`中；

  * 通过将作用域字典作为命名空间来对`code`进行求值，将结果转换为字符串返回，如果成功的话。字段就是个表达式，一切正常。否则(也就是引发了`SyntaxError`异常)，跳到下一步；

  * 执行在相同命名空间(作用域字典)内的字段来对表达式求值，返回空字符串(因为赋值语句没有任何内容进行求值)。

☑ 使用`fileinput`读取所有可用的行，将其放入列表，组合成一个大字符串。

☑ 将所有`field_pat`的匹配项用`re.sub`中的替换函数进行替换，并且打印结果。

注：在之前的*Python*中，将所有行放入列表，最后再联合要比下面这种方法更有效率：

```
text = ""
for line in fileinput.input():
    text += line
```

尽管看起来很优雅，但是每个赋值语句都要创建新的字符串，由旧的字符串和新增加字符串联结在一起组成，这样就会造成严重的资源浪费，使程序运行缓慢。在旧版本的*Python*中，使用*`join`*方法和上述做法之间的差异是巨大的。但是在最近的版本中，使用`+=`运算符事实上会更快。如果觉得性能很重要，那么你可以尝试这两种方式。同时，如果需要一种更优雅的方式来读取文件的所有文本，那么请参见第十一章。

好了，我只用15行代码(不包括空行和注释)就创建了一个强大的模板系统。希望读者已经认识到：使用标准库的时候，Python有多么强大。下面，我们通过测试这个模板系统来结束本例。试着对代码清单10-12中的示例文本运行该系统。

```
# 代码清单10-12 简单的模板示例
[x = 2]
[y = 3]
The sum of [x] and [y] is [x + y].
```

应该会看到如下结果：

```
The sum of 2 and 3 is 5.
```

注：虽然看起来不明显，但是上面的输出包含了*3*个空行——两个在文本上方，一个在下方。尽管前两个字段已经被替换为空字符串，但是随后的空行还留在那里。同时，*`print`*语句增加了新行，也就是末尾的空行。

但是等等，它还能更好！因为使用了`fileinput`，我可以轮流处理几个文件。这意味着可以使用一个文件为变量定义值，而另一个文件作为插入这些值的模板。比如，代码清单10-13包含了定义文件，名为`magnus.txt`，而代码清单10-14则是模板文件，名为`template.txt`。

```
# 代码清单 10-13 一些模板定义
[name     = "Magnus Lie Hetland"]
[email = "magnus@foo.bar"]
[language = "python"]
# 代码清单 10-14 一个模板
[import time]
Dear [name].

I would like to learn how to program. I hear you use
the [language] language a lot -- is it something I should consider?

And, by the way, is [email] your correct email address?

Fooville, [time.asctime()]

Oscar Frozzbozz
```

`import time` 并不是赋值语句(而是准备处理的语句类型)，但是因为我不是过分挑剔的人，所以只用了 `try/except` 语句，使得程序支持任何可以配合 `eval` 或 `exec` 使用的语句和表达式。可以像下面这样运行程序(在UNIX命令行下)：

```
$ python templates.py magnus.txt template.txt
```

你将会看到类似以下内容的输出：

```
Dear Magnus Lie Hetland.

I would like to learn how to program. I hear you use
the python language a lot -- is it something I should consider?

And, by the way, is magnus@foo.bar your correct email address?

Fooville, Wed May 18 20:58:58 2016 Oscar Frozzbozz
```

尽管这个模板系统可以进行功能非常强大的替换，但它还是有些瑕疵的。比如，如果能够使用更灵活的方式来编写定义文件就更好了。如果使用 `execfile` 来执行文件，就可以使用正常的Python语法了。这样也会解决输出内容中顶部出现空行的问题。

还能想到其他改进的方法吗？对于程序中使用的概念，还能想到其他用途吗？精通任何程序设计语言的最佳方法是实践——测试它的限制，探索它的威力。看看你能不能重写这个程序，让它工作得更好并且更能满足需求。

注：事实上，在标准库的 *`string`* 模块中已经有一个非常完美的模板系统了。例如，你可以了解一下 *`Template`* 类。

10.3.9 其他有趣的标准模块

尽管本章内容已经涵盖了很多模块，但是对于整个标准库来说这只是冰山一角。为了引导你进行深入探索，下面会快速介绍一些很酷的库。

☑ `functools` ：你可以从这个库找到一些功能，让你能够通过部分参数来使用某个参数(部分求值)，稍后再为剩下的参数提供数值。在Python3.0中， `filter` 和 `reduce` 包含在该模块中。

☑ `difflib` :这个库让你可以计算两个序列的相似度。还能让你从一些序列中(可供选择的序列列表)找出提供的原始序列“最像”的那个。 `difflib` 可以用于创建简单的搜索程序。

☑ `hashlib` ：通过这个模块，你可以通过字符串计算小“签名”(数字)。如果为两个不同的字符串计算出了签名，几乎可以确保这两个签名完全不同。该模块可以应用与大文本文件，同时在加密和安全性(另见 `md5` 和 `sha` 模块)方面有很多用途。

☑ `csv` ：CSV是逗号分隔值(Comma-Separated Values)的简写，这是一种很多程序(比如很多电子表格和数据库程序)都可以用来存储表格式数据的简单格式。它主要用于在不同程序间交换数据。使用 `csv` 模块可以轻松读写CSV文件，同时以显而易见的方式来处理这种格式的某些很难处理的地方。

☑ `timeit` 、 `profile` 和 `trace` ： `time` 模块(以及它的命令行脚本)是衡量代码片段运行时间的工具。它有很多神秘的功能，你应该用它来代替 `time` 模块进行性能测试。 `profil` e模块(和伴随模块pstats)可用于代码片段效率的全面分析。 `trace` 模块(和程序)可以提供总的分析(也是代码哪部分执行了，哪部分没执行)。这在写测试代码的时候很有用。

☑ `datetime` ：如果 `time` 模块不能满足时间追踪方面的需求，那么 `datetime` 可能就有用武之地了。它支持特殊的日期和时间对象，让你能够以多种方式对它们进行构建和联合。它的接口在很多方面比 `time` 的接口要更加直观。

☑ `itertools` ：它有很多工具用来创建和联合迭代器(或者其他可迭代对象)，还包括实现以下功能的函数：将可迭代的对象链接起来、创建返回无限连续整数的迭代器(和 `range` 类似，但是没有上限)，从而通过重复访问可迭代对象进行循环等等。

☑ `logging` ：通过简单的 `print` 语句打印出程序的哪些方面很有用。如果希望对程序进行跟踪但又不想打印出太多调试内容，那么就需要将这些信息写入日志文件中了。这个模块提供了一组标准的工具，以便让开发人员管理一个或多个核心的日志文件，同时还对日志信息提供了多层次的优先级。

☑ `getopt` 和 `optparse` ：在UNIX中，命令行程序经常使用不同的选项(option)或者开关(switches)运行(Python解释器就是个典型的例子)。这些信息都可以在 `sys.argv` 中找到，但是自己要正确处理它们就没有这么简单了。针对这个问题， `getopt` 库是个切实可行的解决方案，而 `optparse` 则更新、更强大并且更易用。

☑ `cmd` ：使用这个模块可以编写命令行解释器，就像Python的交互式解释器一样。你可以自定义命令，以便让用户能够通过提示符来执行。也许你还能将它作为程序的用户界面。

10.4 小结

本章讲述了模块的知识：如何创建、如何探究以及如何使用标准Python库中的模块。

☑ 模块：从基本上来说，模块就是子程序，它的主函数则用于定义，包括定义函数、类和变量。如果模块包含测试代码，那么应该将这部分代码放置在检查 `__name__ == '__main__'` 是否为真的if语句中。能够在 `PYTHONPATH` 中找到的模块都可以导入。语句 `import foo` 可以导入存储在 `foo.py` 文件中的模块。

☑ 包：包是包含有其他模块的模块。包是作为包含 `__init__.py` 文件的目录来实现的。

☑ 探究模块：将模块导入交互式编辑器后，可以用很多方法对其进行探究。比如使用 `dir` 检查 `__all__` 变量以及使用 `help` 函数。文档和源码是获取信息和内部机制的极好来源。

☑ 标准库：Python包括了一些模块，总称为标准库。本章讲到了其中的很多模块，以下对其中一部分进行回顾。

```
○ `sys`：通过该模块可以访问到多个和Python解释器联系紧密的变量和函数。
○ `os`：通过该模块可以访问到多个和操作系统联系紧密的变量和函数。
○ `fileinput`：通过该模块可以轻松遍历多个文件和流中所有的行。
○ `sets`、`heapq`和`deque`：这3个模块提供了3个有用的数据结构。集合也以内建的类型`set`存在。
○ `time`：通过该模块可以获取当前时间，并可进行时间日期操作和格式化。
○ `random`：通过该模块中的函数可以产生随机数，从序列中选取随机元素以及打乱列表元素。
○ `shelve`：通过该模块可以创建持续性映射，同时将映射的内容保存在给定文件名的数据库中。
○ `re`：支持正则表达式的模块。
```

如果想要了解更多模块，再次建议你浏览Python类库参考，读起来真的很有意思。

10.4.1 本章的新函数

本章涉及的新函数如表10-11所示。

表10-11 本章的新函数

```
dir(obj)          返回按字母顺序排序的属性名称列表
help([obj])       提供交互式帮助或关于特定对象的交互式帮助信息
reload(module)    返回已经导入模块的重新载入版本，该函数在Python3.0将要被废除
```

10.4.2 接下来学什么

如果读者能够掌握本章某些概念，那么你的Python编程水平就会有很大程度的提高。使用手头上的标准库可以让Python从强大变得无比强大。以目前学到的知识为基础，读者已经能编写出用于解决很多问题的程序了。下一章将会介绍如何使用Python和外部世界——文件以及网络——进行交互，从而让读者能够解决更多问题。

第十一章 文件和流

> 来源：http://www.cnblogs.com/Marlowes/p/5519591.html
>
> 作者：Marlowes

到目前为止，本书介绍过的内容都是和解释器自带的数据结构打交道。我们的程序与外部的交互只是通过 `input` 、 `raw_input` 和 `print` 函数，与外部的交互很少。本章将更进一步，让程序能接触更多领域：文件和流。本章介绍的函数和对象可以让你在程序调用时存储数据，并且可以处理来自其他程序的数据。

11.1 打开文件

`open` 函数用来打开文件，语法如下：

```
open(name[, mode[, buffering]])
```

`open` 函数使用一个文件名作为唯一的强制参数，然后返回一个文件对象。模式(`mode`)和缓冲(`buffering`)参数都是可选的，我会在后面的内容中对它们进行解释。

因此，假设有一个名为 `somefile.txt` 的文本文件(可能是用文本编辑器创建的)，其存储路径是 `c:\text` (或者在UNIX下的 `~/text`)，那么可以像下面这样打开文件。

```
>>> f = open(r"C:\text\somefile.txt")
```

如果文件不存在，则会看到一个类似下面这样的异常回溯：

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  IOError: [Errno 2] No such file or directory: 'C:\\text\\somefile.txt'
```

稍后会介绍文件对象的用处。在此之前，先来看看 `open` 函数的其他两个参数。

11.1.1 文件模式

如果 `open` 函数只带一个文件名参数，那么我们可以获得能读取文件内容的文件对象。如果要向文件内写入内容，则必须提供一个模式参数(稍后会具体地说明读和写方式)来显式声明。

`open` 函数中的模式参数只有几个值，如表11-1所示。

明确地指出读模式和什么模式参数都不用的效果是一样的。使用写模式可以向文件写入内容。

`'+'` 参数可以用到其他任何模式中，指明读和写都是允许的。比如 `'r+'` 能在打开一个文本文件用来读写时使用(也可以使用seek方法来实现，请参见本章后面的"随机访问"部分)。

表11-1 `open` 函数中模式参数的常用值

```
'r'                    读模式
'w'                    写模式
'a'                    追加模式
'b'                    二进制模式(可添加到其他模式中使用)
'+'                    读/写模式(可添加到其他模式中使用)
```

`'b'` 模式改变处理文件的方法。一般来说，Python假定处理的是文本文件(包含字符)。通常这样做不会有任何问题。但是如果处理的是一些其他类型的文件(二进制文件)，比如声音剪辑或者图像，那么应该在模式中增加 `'b'` 。参数 `'rb'` 可以用来读取一个二进制文件。

为什么使用二进制模式

如果使用二进制模式来读取(写入)文件的话，与使用文本模式不会有很大区别。仍然能读一定数量的字节(基本上和字符一样)，并且能执行和文本文件有关的操作。关键是，在使用二进制模式时，Python会原样给出文件中的内容——在文本模式下则不一定。

Python对于文本文件的操作方式令人有些惊讶，但不必担心。其中唯一要用到的技巧就是标准化换行符。一般来说，在Python中，换行符(`\n`)表示结束一行并另起一行，这也是UNIX系统中的规范。但在Windows中一行结束的标志是 `\r\n` 。为了在程序中隐藏这些区别(这样的程序就能跨平台运行)，Python在这里做了一些自动转换：当在Windows下用文本模式读取文件中的文本时，Python将 `\r\n` 转换成 `\n` 。相反地，当在Windows下用文本模式向文件写文本时，Python会把 `\n` 转换成 `\r\n` (Macintosh系统上的处理也是如此，只是转换是在 `\r` 和 `\n` 之间进行)。

在使用二进制文件(比如声音剪辑)时可能会产生问题，因为文件中可能包含能被解释成前面提及的换行符的字符，而使用文本模式，Python能自动转换。但是这样会破坏二进制数据。因此为了避免这样的事发生，要使用二进制模式，这样就不会发生转换了。

需要注意的是，在UNIX这种以换行符为标准行结束标志的平台上，这个区别不是很重要，因为不会发生任何转换。

注：通过在模式参数中使用 *`U`* 参数能够在打开文件时使用通用的换行符支持模式，在这种模式下，所有的换行符/字符串(*`\r\n`* 、 *`\r`* 或者是 *`\n`*)都被转换成 *`\n`* ，而不用考虑运行的平台。

11.1.2 缓冲

`open` 函数的第3个参数(可选)控制着文件的缓冲。如果参数是 `0` (或者是 `False`)，I/O(输入/输出)就是无缓冲的(所有的读写操作都直接针对硬盘)；如果是1(或者 `True`)，I/O就是有缓冲的(意味着Python使用内存来代替硬盘，让程序更快，只有使用 `flush` 或者 `close` 时才会更新

硬盘上的数据——参见11.2.4节)。大于1的数字代表缓冲区的大小(单位是字节)，`-1`(或者是任何负数)代表使用默认的缓冲区大小。

11.2 基本的文件方法

打开文件的方法已经介绍了，那么下一步就是用它们做些有用的事情。接下来会介绍文件对象(和一些类文件对象，有时称为流)的一些基本方法。

注：你可能会在*Python*的职业生涯多次遇到类文件这个术语*(*我已经使用了好几次了*)*。类文件对象是支持一些`file`类方法的对象，最重要的是支持`read`方法或者`write`方法，或者两者兼有。那些由`urllib.urlopen`*(*参见第*14*章*)*返回的对象是一个很好的例子。它们支持的方法有`read`、`readline`和`readlines`。但*(*在本书写作期间*)*也有一些方法不支持，如`isatty`方法。

三种标准的流

第10章中关于`sys`模块的部分曾经提到过3种流。它们实际上是文件(或者是类文件对象)：大部分文件对象可用的操作它们也可以使用。

数据输入的标准源是`sys.stdin`。当程序从标准输入读取数据时，你可以通过输入或者使用管道把它和其他程序的标准输出链接起来提供文本(管道是标准的UNIX概念)。

要打印的文本保存在`sys.stdout`内。`input`和`raw_input`函数的提示文字也是写入在`sys.stdout`中的。写入`sys.stdout`的数据一般是出现在屏幕上，但也能使用管道连接到其他程序的标准输入。

错误信息(如栈追踪)被写入`sys.stderr`。它和`sys.stdout`在很多方面都很像。

11.2.1 读和写

文件(或流)最重要的能力是提供或者接受数据。如果有一个名为f的类文件对象，那么就可以用`f.write`方法和`f.read`方法(以字符串形式)写入和读取数据。

每次调用`f.write(string)`时，所提供的参数`string`会被追加到文件中已存在部分的后面。

```
>>> f = open("somefile.txt", "w")
>>> f.write("Hello, ")
>>> f.write("World!") >>> f.close()
```

在完成了对一个文件的操作时，调用`close`。这个方法会在11.2.4节进行详细的介绍。

读取很简单，只要记得告诉流要读多少字符(字节)即可。例子(接上例)如下：

```
>>> f = open("somefile.txt", "r")
>>> f.read(4) 'Hell'
>>> f.read()
'o, World!'
```

首先指定了我要读取的字符数 `"4"` ，然后(通过不提供要读取的字符数的方式)读取了剩下的文件。注意，在调用 `open` 时可以省略模式，因为 `'r'` 是默认的。

11.2.2 管式输出

在UNIX的shell(就像GUN bash)中，使用管道可以在一个命令后面续写其他的多个命令，就像下面这个例子(假设是GUN bash)。

```
$ cat somefile.txt | python somescript.py | sort
```

注：*GUN bash*在*Windows*中也是存在的。 *http://www.cygwin.com* 上面有更多的信息。在*Mac OS X*中，是通过*Terminal*程序，可以使用*shell*文件。

这个管道由以下三3个命令组成。

☑ `cat somefile.txt` ：只是把 `somefile.txt` 的内容写到标准输出(`sys.stdout`)。

☑ `python somescript.py` ：这个命令运行了Python脚本 `somescript` 。脚本应该是从标准输入读，把结果写入到标准输出。

☑ `sort` ：这条命令从标准输入(`sys.stdin`)读取所有的文本，按字母排序，然后把结果写入标准输出。

但管道符号(`|`)的作用是什么？ `somescript.py` 的作用又是什么呢？

管道符号讲一个命令的标准输出和下一个命令的标准输入连接在一起。明白了吗？这样，就知道 `somescript.py` 会从它的 `sys.stdin` 中读取数据(`cat somefile.txt` 写入的)，并把结果写入它的 `sys.stdout` (`sort` 在此得到数据)中 。

使用 `sys.stdin` 的一个简单的脚本(`somescript`)如代码清单11-1所示。 `somefile.txt` 文件的内容如代码清单11-2所示。

```
# 代码清单 11-1 统计`sys.stdin`中单词数的简单脚本
# somescript.py

import sys
text = sys.stdin.read()
words = text.split()
wordcount = len(words)
print "Wordcount:", wordcount
# 代码清单 11-2 包含示例文本的文件
Your mother was a hamster and your father smelled of elderberries.
```

下面是 `cat somefile.txt | python somescript.py` 的结果。

```
Wordcount: 11
```

随机访问

本章内的例子把文件都当成流来操作，也就是说只能按照从头到尾的顺序读数据。实际上，在文件中随意移动读取位置也是可以的，可以使用类文件对象的方法`seek`和`tell`来直接访问感兴趣的部分(这种做法称为随机访问)。

```
seek(offset[, whence])
```

这个方法把当前位置(进行读和写的位置)移动到由`offset`和`whence`定义的位置。`Offset`类是一个字节(字符)数，表示偏移量。`whence`默认是0，表示偏移量是从文件开头开始计算的(偏移量必须是非负的)。`whence`可能被设置为1(相对于当前位置的移动，此时偏移量`offset``可以是负的)或者2(相对于文件结尾的移动)。

考虑下面这个例子：

```
>>> f = open(r"c:\text\somefile.txt", "w")
>>> f.write("01234567890123456789")
>>> f.seek(5)
>>> f.write("Hello, World!")
>>> f.close()
>>> f = open(r"c:\text\somefile.txt")
>>> f.read()
>>> '01234Hello, World!89'
# tell方法返回当前文件的位置如下例所示：
>>> f = open(r"c:\text\somefile.txt")
>>> f.read(3)
>>> '012'
>>> f.read(2)
>>> '34'
>>> f.tell()
>>> 5L
```

11.2.3 读写行

实际上，程序到现在做的工作都是很不实用的。通常来说，逐个字符串读取文件也是没问题的，进行逐行的读取也可以。还可以使用file.readline读取单独的一行(从当前位置开始直到一个换行符出现，也读取这个换行符)。不使用任何参数(这样，一行就被读取和返回)或者使用一个非负数的整数作为`readline`可以读取的字符(或字节)的最大值。因此，如果`someFile.readline()`返回`"Hello, World!\n"`，`someFile.readline(5)`返回`"Hello"`。`readlines`方法可以读取一个文件中的所有行并将其作为列表返回。

`writelines`方法和`readlines`相反：传给它一个字符串的列表(实际上任何序列或者可迭代的对象都行)，它会把所有的字符串写入文件(或流)。注意，程序不会增加新行，需要自己添加。没有`writeline`方法，因为能使用`write`。

注：在使用其他的符号作为换行符的平台上，用 `\r` *(Mac*中*)*和 `\r\n` *(Windows*中*)*代替 `\n` *(*有 `os.linesep` 决定*)*。

11.2.4 关闭文件

应该牢记使用 `close` 方法关闭文件。通常来说，一个文件对象在退出程序后(也可能在退出前)自动关闭，尽管是否关闭文件不是很重要，但关闭文件是没有什么害处的，可以避免在某些操作系统或设置中进行无用的修改，这样做也会避免用完系统中所打开文件的配额。

写入过的文件总是应该关闭，是因为Python可能会缓存(出于效率的考虑而把数据临时地存储在某处)写入的数据，如果程序因为某些原因崩溃了，那么数据根本就不会被写入文件。为了安全起见，要在使用完文件后关闭。

如果想确保文件被关闭了，那么应该使用 `try/finally` 语句，并且在 `finally` 子句中调用 `close` 方法。

```
# Open your file here
try:
    # Write data to your file
finally:
    file.close()
```

事实上，有专门为这种情况设计的语句(在Python2.5中引入)，即 `with` 语句：

```
with open("somefile.txt") as somefile:
    do_something(somefile)
```

`with` 语句可以打开文件并且将其赋值到变量上(本例是 `somefile`)。之后就可以将数据写入语句体中的文件(或许执行其他操作)。文件在语句结束后会被自动关闭，即使是处于异常引起的结束也是如此。

在Python2.5中， `with` 语句只有在导入如下的模块后才可以用：

```
from __future__ import with_statement
```

而2.5之后的版本中， `with` 语句可以直接使用。

注：在写入了一些文件的内容后，通常的想法是希望这些改变会立刻体现在文件中，这样一来其他读取这个文件的程序也能知道这个改变。哦，难道不是这样吗？不一定。数据可能被缓存了(在内存中临时性地存储)，直到关闭文件才会被写入到文件。如果需要继续使用文件(不关闭文件)，又想将磁盘上的文件进行更新，以反映这些修改，那么就要调用文件对象的 `flush` 方法(注意， `flush` 方法不允许其他程序使用该文件的同时访问文件，具体的情况依据使用的操作系统和设置而定。不管在什么时候，能关闭文件时最好关闭文件)。

上下文管理器

`with` 语句实际上是很通用的结构，允许使用所谓的上下文管理器(context manager)。上下文管理器是一种支持 `__enter__` 和 `__exit__` 这两个方法的对象。

`__enter__` 方法不带参数，它在进入 `with` 语句块的时候被调用，返回值绑定到在 `as` 关键字之后的变量。

`__exit__` 方法带有3个参数：异常类型、异常对象和异常回溯。在离开方法(通过带有参数提供的、可引发的异常)时这个函数被调用。如果 `__exit__` 返回 `false` ，那么所有的异常都不会被处理。

文件可以被用作上下文管理器。它们的 `__enter__` 方法返回文件对象本身， `__exit__` 方法关闭文件。有关这个强大且高级的特性的更多信息，请参看Python参考手册中的上下文管理器部分。或者可以在Python库参考中查看上下文管理器和 `contextlib` 部分。

11.2.5 使用基本文件方法

假设 `somefile.txt` 包含如代码清单11-3所示的内容，能对它进行什么操作？

```
# 代码清单11-3 一个简单的文本文件
Welcome to this file
There is nothing here except This stupid haiku
```

让我们试试已经知道的方法，首先是 `read(n)` ：

```
>>> f = open(r"C:\text\somefile.txt")
>>> f.read(7) 'Welcome'
>>> f.read(4) ' to '
>>> f.close()
```

然后是 `read()` ：

```
>>> f = open(r"C:\text\somefile.txt")
>>> print f.read()
Welcome to this file
There is nothing here except This stupid haiku
>>> f.close()
```

接着是 `readline()` ：

```
>>> f = open(r"C:\text\somefile.txt")
>>> for i in range(3):
...     print str(i) + ": " + f.readline(),
...
0: Welcome to this file
1: There is nothing here except
2: This stupid haiku
>>> f.close()
```

以及 `readlines()` ：

```
>>> import pprint
>>> pprint.pprint(open(r"C:\text\somefile.txt").readlines())
['Welcome to this file\n', 'There is nothing here except\n', 'This stupid haiku']
```

注意，本例中我所使用的是文件对象自动关闭的方式。

下面是写文件，首先是`write(string)`：

```
>>> f = open(r"C:\text\somefile.txt", "w")
>>> f.write("this\nis no\nhaiku")
>>> f.close()
```

在运行这个程序后，文件包含的内容如代码清单11-4所示。

```
# 代码清单11-4 修改了的文本文件
this is no
haiku
```

最后是`writelines(list)`：

```
>>> f = open(r"C:\text\somefile.txt")
>>> lines = f.readlines()
>>> f.close()
>>> lines[1] = "isn't a\n"
>>> f = open(r"C:\text\somefile.txt", "w")
>>> f.writelines(lines)
>>> f.close()
```

运行这个程序后，文件包含的文本如代码清单11-5所示。

```
# 代码清单11-5 再次修改的文本文件
this
isn't a
haiku
```

11.3 对文件内容进行迭代

前面介绍了文件对象提供的一些方法，以及如何获取这样的文件对象。对文件内容进行迭代以及重复执行一些操作，是最常见的文件操作之一。尽管有很多方法可以实现这个功能，或者可能有人会偏爱某一种并坚持只使用那种方法，但是还有一些人使用其他的方法，为了能理解他们的程序，你就应该了解所有的基本技术。其中的一些技术是使用曾经见过的方法(如`read`、`readline`和`readlines`)，另一些方法是我即将介绍的(比如`xreadlines`和文件迭代器)。

在这部分的所有例子中都使用了一个名为`process`的函数，用来表示每个字符或每行的处理过程。读者也可以用你喜欢的方法自行实现这个函数。下面就是一个例子：

```
def process(string):
    print "Processing: ", string
```

更有用的实现是在数据结构中存储数据，计算和值，用 `re` 模块来代替模式或者增加行号。

如果要尝试实现以上功能，则应该把 `filename` 变量设置为一个实际的文件名。

11.3.1 按字节处理

最常见的对文件内容进行迭代的方法是在 `while` 循环中使用 `read` 方法。例如，对每个字符(字节)进行循环，可以用代码清单11-6所示的方法实现。

```
# 代码清单11-6 用read方法对每个字符进行循环
f = open(filename)
char = f.read(1)
while char:
    process(char)
    char = f.read(1)
f.close()
```

这个程序可以使用是因为当到达文件的末尾时， `read` 方法返回一个空的字符串，但在那之前返回的字符串会包含一个字符(这样布尔值是真)。如果 `char` 是真，则表示还没有到文件末尾。

可以看到，赋值语句 `char = f.read(1)` 被重复地使用，代码重复通常被认为是一件坏事。(懒惰是美德，还记得吗？)为了避免发生这种情况，可以使用在第五章介绍过的 `while true/break` 语句。最终的代码如代码清单11-7所示。

```
# 代码清单11-7 用不同的方式写循环
f = open(filename)
while True:
    char = f.read()
    if not char:
    break
    process(char)
f.close
```

如在第五章提到的， `break` 语句不应该频繁地使用(因为这样会让代码很难懂)；尽管如此，代码清单11-7中使用的方法比代码清单11-6中的方法要好，因为前者避免了重复的代码。

11.3.2 按行操作

当处理文本文件时，经常会对文件的行进行迭代而不是处理单个字符。处理行使用的方法和处理字符一样，即使用 `readline` 方法(先前在11.2.3节介绍过)，如代码清单11-8所示。

```
# 代码清单11-8 在while循环中使用readline
f = open(filename)
while True:
    line = f.readline()
    if not line:
    break
    process(line)
f.close()
```

11.3.3 读取所有内容

如果文件不是很大，那么可以使用不带参数的 `read` 方法一次读取整个文件(把整个文件当做一个字符串来读取)，或者使用 `readlines` 方法(把文件读入一个字符串列表，在列表中每个字符串就是一行)。代码清单11-9和代码清单11-10展示了在读取这样的文件时，在字符串和行上进行迭代是多么容易。注意，将文件的内容读入一个字符串或者是读入列表在其他时候也很有用。比如在读取后，就可以对字符串使用正则表达式操作，也可以将行列表存入一些数据结构中，以备将来使用。

```
# 代码清单11-9 用read迭代每个字符
f = open(filename)
for char in f.read():
    process(char)
f.close()
# 代码清单11-10 用readlines迭代行
f = open(filename)
for line in f.readlines():
    process(line)
f.close()
```

11.3.4 使用 `fileinput` 实现懒惰行迭代

在需要对一个非常大的文件进行行迭代的操作时， `readlines` 会占用太多的内存。这个时候可以使用 `while` 循环和 `readline` 方法来替代。当然，在Python中如果能使用 `for` 循环，那么它就是首选。本例恰好可以使用 `for` 循环可以使用一个名为懒惰行迭代的方法：说它懒惰是因为它只是读取实际需要的文件部分。

第十章内已经介绍过 `fileinput` ，代码清单11-11演示了它的用法。注意， `fileinput` 模块包含了打开文件的函数，只需要传一个文件名给它。

```
# 代码清单11-11 用fileinput来对行进行迭代

import fileinput
for line in fileinput.input(filename):
    process(line)
```

注：在旧式代码中，可使用 *`xreadlines`* 实现懒惰行迭代。它的工作方式和 *`readlines`* 很类似，不同点在于，它不是将全部的行读到列表中而是创建了一个 *`xreadlines`* 对象。注意， *`xreadlines`* 是旧式的，在你自己的代码中最好用 *`fileinput`* 或文件迭代器*(*下面来介绍*)*。

11.3.5 文件迭代器

现在是展示所有最酷的技术的时候了，在Python中如果一开始就存在这个特性的话，其他很多方法(至少包括 `xreadlines`)可能就不会出现了。那么这种技术到底是什么？在Python的近几个版本中(从2.2开始)，文件对象是可迭代的，这就意味着可以直接在 `for` 循环中使用它们，从而对它们进行迭代。如代码清单11-12所示，很优雅，不是吗？

```
# 代码清单11-12 迭代文件
f = open(filename) for line in f:
    process(line)
f.close()
```

在这些迭代的例子中，都没有显式的关闭文件的操作，尽管在使用完以后，文件的确应该关闭，但是只要没有向文件内写入内容，那么不关闭文件也是可以的。如果希望由Python来负责关闭文件(也就是刚才所做的)，那么例子应该进一步简化，如代码清单11-13所示。在那个例子中并没有把一个打开的文件赋给变量(就像我在其他例子中使用的变量 `f`)，因此也就没办法显式地关闭文件。

```
# 代码清单11-13 对文件进行迭代而不使用变量存储文件对象

for line in open(filename):
    process(line)
```

注意 `sys.stdin` 是可迭代的，就像其他的文件对象。因此如果想要迭代标准输入中的所有行，可以按如下形式使用 `sys.stdin` 。

```
import sys
for line in sys.stdin:
    process(line)
```

可以对文件迭代器执行和普通迭代器相同的操作。比如将它们转换为字符串列表(使用list(open(filename)))，这样所达到的效果和使用readlines一样。

考虑下面的例子：

```
>>> f = open("somefile.txt", "w")
>>> f.write("First line\n")
>>> f.write("Second line\n")
>>> f.write("Third line\n")
>>> f.close()
>>> lines = list(open("somefile.txt"))
>>> lines
['First line\n', 'Second line\n', 'Third line\n']
>>> first, second, third = open("somefile.txt")
>>> first 'First line\n'
>>> second 'Second line\n'
>>> third 'Third line\n'
```

在这个例子中，注意下面的几点很重要。

☑ 在使用`print`来向文件内写入内容，这会在提供的字符串后面增加新的行。

☑ 使用序列来对一个打开的文件进行解包操作，把每行都放入一个单独的变量中(这么做是很有实用性的，因为一般不知道文件中有多少行，但它演示了文件对象的"迭代性")。

☑ 在写文件后关闭了文件，是为了确保数据被更新到硬盘(你也看到了，在读取文件后没有关闭文件，或许是太马虎了，但并没有错)。

11.4 小结

本章中介绍了如何通过文件对象和类文件对象与环境互动，I/O也是Python中最重要的技术之一。下面是本章的关键知识。

☑ 类文件对象：类文件对象是支持`read`和`readline`方法(可能是`write`和`writelines`)的非正式对象。

☑ 打开和关闭文件：通过提供一个文件名，使用`open`函数打开一个文件(在新版的Python中实际上是`file`的别名)。如果希望确保文件被正常关闭，即使发生错误时也是如此可以使用`with`语句。

☑ 模式和文件类型：当打开一个文件时，也可以提供一个模式，比如`'r'`代表读模式，`'w'`代表写模式。还可以将文件作为二进制文件打开(这个只在Python进行换行符转换的平台上才需要，比如Windows，或许其他地方也应该如此)。

☑ 标准流：3个标准文件对象(在`sys`模块中的`stdin`、`stdout`和`stderr`)是一个类文件对象，该对象实现了UNIX标准的I/O机制(Windows中也能用)。

☑ 读和写：使用`read`或是`write`方法可以对文件对象或类文件对象进行读写操作。

☑ 读写行：使用`readline`和`readlines`和(用于有效迭代的)`xreadlines`方法可以从文件中读取行，使用`writelines`可以写入数据。

☑ 迭代文件内容：有很多方法可以迭代文件的内容。一般是迭代文本中的行，通过迭代文件对象本身可以轻松完成，也有其他的方法，就像`readlines`和`xreadlines`这两个倩兼容Python老版本的方法。

11.4.1 本章的新函数

本章涉及的新函数如表11-2所示。

表11-2 本章的新函数

```
file(name[, mode[, buffering]])  打开一个文件并返回一个文件对象
open(name[, mode[, buffering]])  file的别名；在打开文件时，使用open而不是file
```

11.4.2 接下来学什么

现在你已经知道了如何通过文件与环境交互，但怎么和用户交互呢？到现在为止，程序已经使用的只有`input`、`raw_input`和`print`函数，除非用户在程序能够读取的文件中写入一些内容，否则没有任何其他工具能创建用户界面。下一章会介绍图形用户界面(graphical user interface)中的窗口、按钮等。

第十二章 图形用户界面

> 来源：http://www.cnblogs.com/Marlowes/p/5520948.html
>
> 作者：Marlowes

本章将会介绍如何创建Python程序的图形用户界面(GUI)，也就是那些带有按钮和文本框的窗口等。很酷吧？

目前支持Python的所谓“GUI工具包”(GUI Toolkit)有很多，但是没有一个被认为是标注你的GUI工具包。这样的情况也好(自由选择的空间较大)也不好(其他人没法用程序，除非他们也安装了相同的GUI工具包)，幸好Python的GUI工具包之间没有冲突，想装多少个就可以装多少个。

本章简要介绍最成熟的跨平台Python GUI工具包——wxPython。有关更多wxPython程序的介绍，请参考官方文档。关于GUI程序设计的更多信息请参见第二十八章。

12.1 丰富的平台

在编写Python GUI程序前，需要决定使用哪个GUI平台。简单来说，平台是图形组件的一个特定集合，可以通过叫做GUI工具包的给定Python模块进行访问。Python可用的工具包很多。其中一些最流行的如表12-1所示。要获取更加详细的列表，可以在Vaults of Parnassus上面以关键字"GUI"进行搜索。也可以在Python Wiki上找到完全的工具列表。Guiherme Polo也撰写过一篇有关4个主要平台对比的论文("PyGTK,PyQt,Tkinter and wxPython comparison" (PyGTK、PyQt、Tkinter和wxPython的比较)，The Python Papers， 卷3，第1期26~37页 。这篇文章可以从 http://pythonpapers.org 上获得)。

表12-1 一些支持Python的流行GUI工具包

```
Tkinter         使用Tk平台，很容易得到，半标准。                    http://wiki.python.or
g/moin/TkInter
wxpython        基于wxWindows，跨平台越来越流行。                  http://wxpython.org
PythonWin       只能在Windows上使用，使用了本机的Windows GUI功能    http://starship.python.
net/crew/mhammond
Java Swing      只能用于Jython，使用本机的Java GUI。               http://java.sun.com/d
ocs/books/tutorial/uiswing
PyGTK           使用GTK平台，在Linux上很流行。                     http://pygtk.org
PyQt            使用Qt平台，跨平台。                               http://wiki.python.org/mo
in/PyQt
```

可选的包太多了，那么应该用哪个呢？尽管每个工具包都有利弊，但很大程度上取决于个人喜好。Tkinter实际上类似于标准，因为它被用于大多数“正式的”Python GUI程序，而且它是Windows二进制发布版的一部分。但是在UNIX上要自己编译安装。Tkinter和Swing Jython将在12.4节进行介绍。

另外一个越来越受欢迎的工具是wxPython。这是个成熟而且特性丰富的包，也是Python之父Guido van Rossum的最爱。在本章的例子中，我们将使用wxPython。

关于Pythonwin、PyGTK和PyQt的更多信息，请查看这些项目的主页(见表12-1)。

12.2 下载和安装wxPython

要下载wxPython，只要访问它的下载页面即可。这个网页提供了关于下载哪个版本的详细指导，还有使用不同版本的先决条件。

如果使用Windows系统，应该下载预建的二进制版本。可以选择支持Unicode或不支持Unicode的版本，除非要用到Unicode，否则选择哪个版本区别并不大。确保所选择的二进制版本要对应Python的版本。例如，针对Python2.3进行编译的wxPython并不能用于Python2.4。

对于Mac OS X来说，也应该选择对应Python版本的wxPython。可能还需要考虑操作系统版本。同样，你也可以选择支持Unicode和不支持Unicode的版本。下载链接和相关的解释能非常明确地告诉你应该下载哪个版本。

如果读者正在使用Linux，那么可以查看包管理器中是否包括wxPython，它存在于很多主流发布版本中。对于不同版本的Linux来说也有不同的RPM包。如果运行包含RPM的Linux发行版，那么至少应该下载wxPython和运行时包(runtime package)，而不需要devel包。再说一次，要选择与Python以及Linux发布版对应的版本。

如果没有任何版本适合硬件或操作系统(或者Python版本)，可以下载源代码发布版。为了编译可能还需要根据各种先决条件下载其他的源代码包，这已经超出了本章的范围。这些内容在wxPython的下载页面上都有详细的解释。

在下载了wxPython之后，强烈建议下载演示版本(demo，它必须进行独立安装)，其中包含文档、示例程序和非常详细的(而且有用的)演示分布。这个演示程序演示了大多数wxPython的特性，并且还能以对用户非常友好的方式查看各部分源代码——如果想要自学wxPython的话非常值得一看。

安装过程应该很简单，而且是自动完成的。安装Windows二进制版本只要运行下载完的可执行文件(.exe文件)；在OS X系统中，下载后的文件应该看起来像是可以打开的CD-ROM一样，并带有一个可以双击的.pkg文件。要使用RPM安装，请参见RPM文档。Windows和Mac OS X版本都会运行一个安装向导，用起来很简单。只要选择默认设置即可，然后一直惦记Continue，最后点击Finish即可。

12.3 创建示例GUI应用程序

为使用wxPython进行演示，首先来看一下如何创建一个简单的示例GUI应用程序。你的任务是编写一个能编辑文本文件的基础程序。编写全功能的文本编辑器已经超出了本章的范围——关注的只是基础。毕竟目标是演示在Python中进行GUI编程的基本原理。

对这个小型文本编辑器的功能要求如下：

☑ 它应允许打开给定文件名的文本文件；

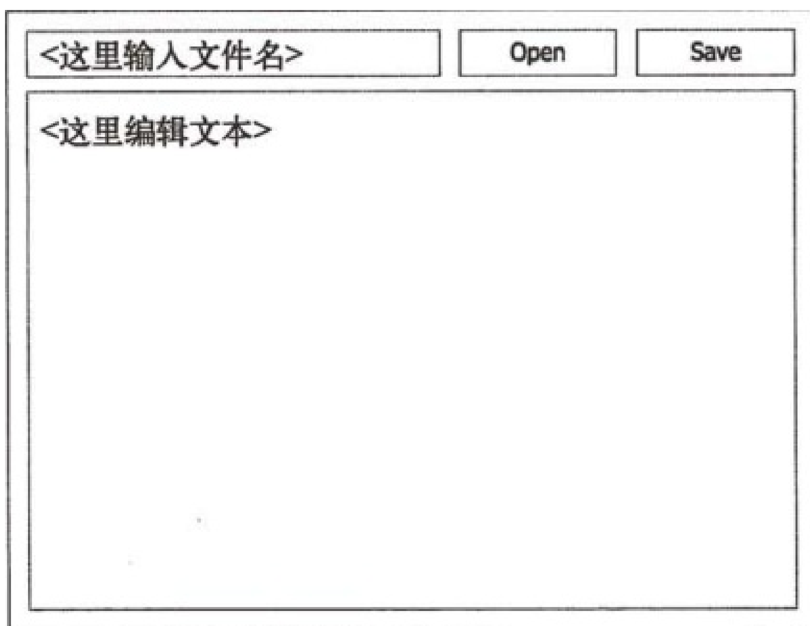
☑ 它应允许编辑文本文件；

☑ 它应允许保存文本文件；

☑ 它应允许退出程序。

当编写GUI程序时，画个界面草图总是有点用的。图12-1展示了一个满足我们文本编辑要求的布局：

图12-1 文本编辑器草图



界面元素可以像下面这样使用。

☑ 在按钮左侧的文本框内输入文件名，点击Open打开文件。文件中包含的文本会显示在下方的文本框内。

☑ 可以在这个大的文本框中随心所欲地编辑文本。

☑ 如果希望保存修改，那么点击Save按钮，会再次用到包含文件名的文本框——然后将文本框的内容写入文件。

☑ 没有Quit(退出)按钮——如果用户关闭窗口，程序就会退出。

使用某些语言写这样的程序时相当难的。但是利用Python和恰当的GUI工具包，简直是小菜一碟(虽然现在读者可能不同意这种说法，但在学习完本章之后应该就会同意了)。

12.3.1 开始

为了查看wxPython是否能工作，可以尝试运行wxPython的演示版本(要单独安装)。Windows内应该可以在开始菜单找到，而OS X可以直接把wxPython Demo文件拖到应用程序中，然后运行。看够了演示就可以开始写自己的程序了，当然，这会更有趣的。

开始需要导入 `wx` 模块：

```
import wx
```

编写wxPython程序的方法很多，但不可避免的事情是创建应用程序对象。基本的应用程序类叫做ex.App，它负责幕后所有的初始化。最简单的wxPython程序应该像下面这样：

```
import wx

app = wx.App()
app.MainLoop()
```

注：如果 `wx.App` 无法工作，可能需要将它替换为 `wx.PySimpleApp` 。

因为没有任何用户可以交互的窗口，程序会立刻退出。

例中可以看到， `wx` 包中的方法都是以大写字母开头的，而这和Python的习惯是相反的。这样做的原因是这些方法名和基础的C++包wxWidgets中的方法名都是对应的。尽管没有正式的规则反对方法或者函数名以大写字母开头，但是规范的做法是为类保留这样的名字。

12.3.2 窗口和组件

窗口(Windows)也成为框架(`frame`)，它只是 `wx.Frame` 类的实例。 `wx` 框架中的部件都是由它们的父部件作为构造函数的第一个参数创建的。如果正在创建一个单独的窗口，就不需要考虑父部件，使用None即可，如代码清单12-1所示。而且在调用 `app.MainLoop` 前需要调用窗口的 `Show` 方法——否则它会一直隐藏(可以在事例处理程序中调用 `win.Show` ，后面会介绍)。

```
# 代码清单12-1 创建并且显示一个框架

import wx

app = wx.App()
win = wx.Frame(None)
win.Show()
app.MainLoop()
```

如果运行这个程序，应该能看到一个窗口出现，类似于图12-2。

图12-2 只有一个窗口的GUI程序



在框架上增加按钮也很简单——只要使用win作为符参数实例化`wx.Button`即可，如代码清单12-2所示。

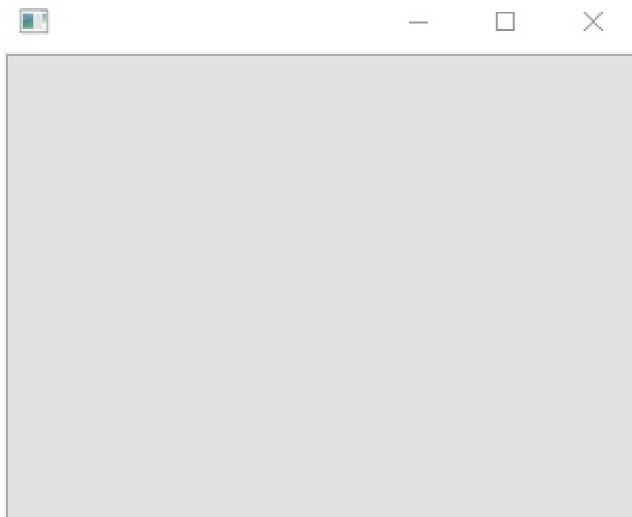
```
# 代码清单12-2 在框架上增加按钮

import wx

app = wx.App()
win = wx.Frame(None)
btn = wx.Button(win)
win.Show()
app.MainLoop()
```

这样会得到一个带有一个按钮的窗口，如图12-3所示。

图12-3 增加按钮后的程序



当然，这里做的还不够，窗口没有标题，按钮没有标签，而且也不希望让按钮覆盖整个窗口。

12.3.3 标签、标题和位置

可以在创建部件的时候使用构造函数的label参数设定它们的标签。同样，也可以用`title`参数设定框架的标题。我发现最实用的做法是为`wx`构造函数使用关键字参数，所以我不用记住参数的顺序。代码清单12-3演示了一个例子。

```
# 代码清单12-3 使用关键字参数增加标签和标题

import wx

app = wx.App()

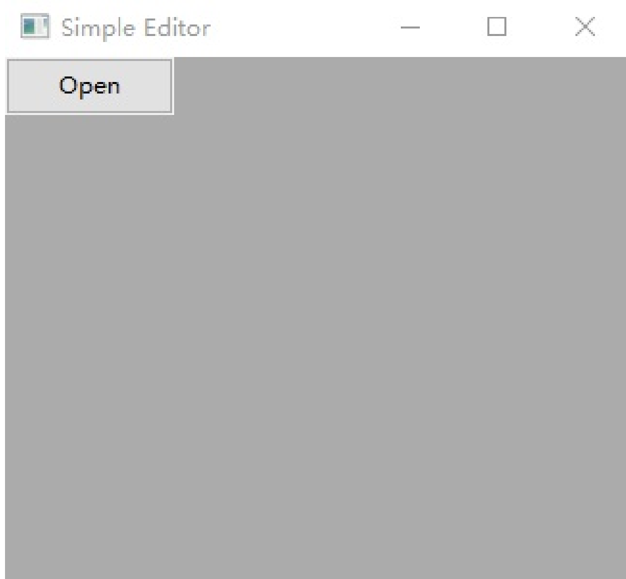
win = wx.Frame(None, title="Simple Editor")
loadButton = wx.Button(win, label="Open")
saveButton = wx.Button(win, label="Save")

win.Show()
app.MainLoop()
```

程序的运行结果如图12-4所示。

这个版本的程序还是有些不对——好像丢了一个按钮！实际上它没丢——只是隐藏了。注意一下按钮的布局就能将隐藏的按钮显示出来。一个很基础(但是不实用)的方法是使用pos和size参数在构造函数内设置位置和尺寸，如代码清单12-4所示。

12-4 有布局问题的窗口



```
# 代码清单12-4 设置按钮位置

import wx

app = wx.App()

win = wx.Frame(None, title="Simple Editor", size=(410, 335))
win.Show()

loadButton = wx.Button(win, label="Open",
                       pos=(225, 5), size=(80, 25))
saveButton = wx.Button(win, label="Save",
                       pos=(315, 5), size=(80, 25))

filename = wx.TextCtrl(win, pos=(5, 5), size=(210, 25))

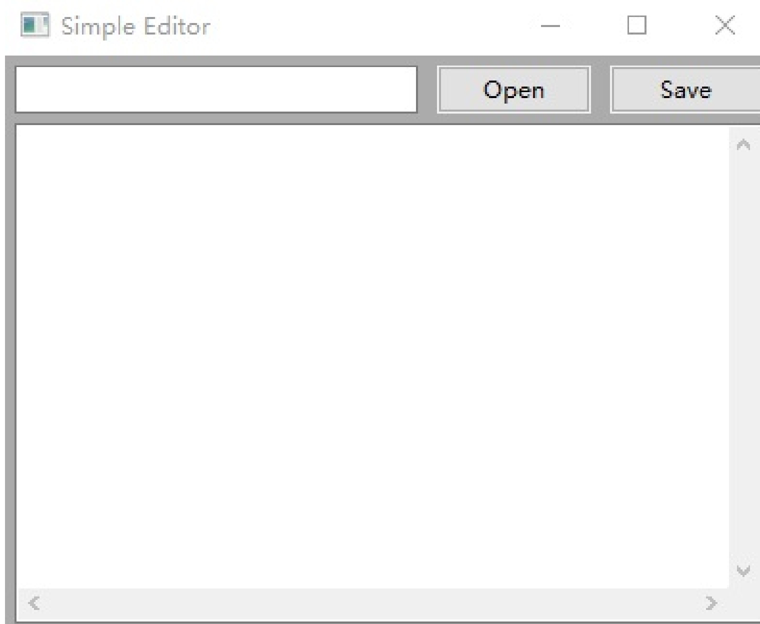
contents = wx.TextCtrl(win, pos=(5, 35), size=(390, 260),
                       style=wx.TE_MULTILINE | wx.HSCROLL)

app.MainLoop()
```

你看到了，位置和尺寸都包括一对数值：位置包括x和y坐标，而尺寸包括宽和高。

这段代码中还有一些新东西：我创建了两个文本控件(`text control` ， `wx.TextCtrl` 对象)，每个都使用了自定义风格。默认的文本控件是文本框(text field)，就是一行可编辑的文本，没有滚动条，为了创建文本区(text area)只要使用 `style` 参数调整风格即可。 `style` 参数的值实际上是个整数，但不用直接指定，可以使用按位或运算符OR(或管道运算符)联合 `wx` 模块中具有特殊名字的风格来指定。本例中，我联合了 `wx.TE_MULTILINE` 来获取多行文本区(默认有垂直滚动条)以及 `wx.HSCROLL` 来获取水平滚动条。程序运行的结果如图12-5所示。

图12-5 位置合适的组件



12.3.4 更智能的布局

尽管明确每个组件的几何位置很容易理解，但是过程很乏味。在绘图纸上画出来可能有助于确定坐标，但是用数字来调整位置的方法有很多严重的缺点。如果运行程序并且试图调整窗口大小，那么会注意到组件的几何位置不变。虽然不是什么大事，但是看起来还是有些奇怪。在调整窗口大小时，应该能保证窗口中的组件也会随之调整大小和位置。

考虑一下我是如何布局的，那么出现这种情况就不会令人惊讶了。每个组件的位置和大小都显式设定的，但是没有明确在窗口大小变化的时候它们的行为是什么。指定行为的方法有很多，在wx内进行布局的最简单方法是使用尺寸器(sizer)，最容易使用的工具就是 `wx.BoxSizer` 。

尺寸器会管理组件的尺寸。只要将部件添加到尺寸器上，再加上一些布局参数，然后让尺寸器自己去管理父组件的尺寸。在刚才的例子中，需要增加背景组件(`wx.Panel`)，创建一些嵌套的 `wx.BoxSizer` ，然后使用面板的 `SetSizer` 方法设定它的尺寸器，如代码清单12-5所示。

```
import wx

app = wx.App()

win = wx.Frame(None, title="Simple Editor", size=(410, 335))
bkg = wx.Panel(win)

loadButton = wx.Button(bkg, label="open")
saveButton = wx.Button(bkg, label="Save")
filename = wx.TextCtrl(bkg)
contents = wx.TextCtrl(bkg, style=wx.TE_MULTILINE | wx.HSCROLL)

hbox = wx.BoxSizer()
hbox.Add(filename, proportion=1, flag=wx.EXPAND)
hbox.Add(loadButton, proportion=0, flag=wx.LEFT, border=5)
hbox.Add(saveButton, proportion=0, flag=wx.LEFT, border=5)

vbox = wx.BoxSizer(wx.VERTICAL)
vbox.Add(hbox, proportion=0, flag=wx.EXPAND | wx.ALL, border=5)
vbox.Add(contents, proportion=1,
         flag=wx.EXPAND | wx.LEFT | wx.BOTTOM | wx.RIGHT, border=5)

bkg.SetSizer(vbox)
win.Show()

app.MainLoop()
```

这段代码的运行结果和前例相同，但是使用了相对坐标而不是绝对坐标。

`wx.BoxSizer` 的构造函数带有一个决定它是水平还是垂直的参数(`wx.HORIZONTAL` 或者 `wx.VERTICAL`)，默认为水平。 `Add` 方法有几个参数， `proportion` 参数根据在窗口改变大小时所分配的空间设置比例。例如，水平的 `BoxSizer` (第一个)中， `filename` 组件在改变大小时获取了全部的额外空间。如果这3个部件都把 `proportion` 设为1，那么都会获得相等的空间。可以将 `proportion` 设定为任何数。

`flag` 参数类似于构造函数中的 `style` 参数，可以使用按位或运算符连接构造符号常量(`symbolic constan` t，即有特殊名字的整数)对其进行构造。 `wx.EXPAND` 标记确保组件会扩展到所分配的空间中。而 `wx.LEFT` 、 `wx.RIGHT` 、 `wx.TOP` 、 `wx.BOTTOM` 和 `wx.ALL` 标记决定边框参数应用于哪个边，边框参数用于设置边缘宽度(间隔)。

就是这样。我得到了我要的布局。但是遗漏了一件至关重要的事情——按下按钮，却什么都没发生。

注：更多有关尺寸器的信息或者与*wxPython*相关的信息请参见*wxPython*的演示版本，它里面会有你想要了解的内容和示例代码。如果看起来比较难，可以访问*wxPython*的网站。

12.3.5 事件处理

在GUI术语中，用户执行的动作(比如点击按钮)叫做事件(event)。你需要让程序注意这些事件并且做出反应。可以将函数绑定到所涉及的时间可能发生的组件上达到这个效果。当事件发生时，函数会被调用。利用部件的Bind方法可以将时间处理函数链接到给定的事件上。

假设写了一个负责打开文件的函数，并将其命名为 `load` 。然后就可以像下面这样将该函数作为loadButton的事件处理函数：

```
loadButton.Bind(wx.EVT_BUTTON, load)
```

很直观，不是吗？我把函数链接到了按钮——点击按钮的时候，函数被调用。名为 `wx.EVT_BUTTON` 的符号常量表示一个按钮事件。 `wx` 框架对于各种事件都有这样的事件常量——从鼠标动作到键盘按键，等等。

为什么用LOAD？

注：我之所以用 *`loadButton`* 和 *`load`* 作为按钮以及处理函数的名字不是偶然的——尽管按钮的文本为 *`"Open"`* 。这是因为如果我把按钮叫做 *`openButton`* 的话， *`open`* 也就自然成了事件处理函数的名字，这样就和内建的文件打开函数 *`open`* 冲突，导致后者失效。虽然有很多种方法可以解决这个问题，但我觉得使用不同的名字是最简单的。

12.3.6 完成了的程序

让我们来完成剩下的工作。现在需要的就是两个事件处理函数： `load` 和 `save` 。当事件处理函数被调用时，它会收到一个事件对象作为它唯一的参数，其中包括发生了什么事情的信息，但是在这里可以忽略这方面的事情，因为程序只关心点击时发生的事情。

```
def load(event):
    file = open(filename.GetValue())
    contents.SetValue(file.read())
    file.close()
```

读过第十一章后，读者应该对于文件打开/读取的部分比较熟悉了。文件名使用 `filename` 对象的 `GetValue` 方法获取(`filename` 是小的文本框)。同样，为了将文本引入文本区，只要使用 `contents.SetValue` 即可。

`save` 函数也很简单：几乎和 `load` 一样——除了它有个 `"w"` 标志，以及用于文件处理部分的 `write` 方法。`GetValue` 用于从文本区获得信息。

```
def save(event):
    file = open(filename.GetValue(), "w")
    file.write(contents.GetValue())
    file.close()
```

就是这样了。现在我将这些函数绑定到相应的按钮上，程序已经可以运行了。最终的程序如代码清单12-6所示。

```
#!/usr/bin/env python # coding=utf-8

import wx def load(event):
    file = open(filename.GetValue())
    contents.SetValue(file.read())
    file.close() def save(event):
    file = open(filename.GetValue(), "w")
    file.write(contents.GetValue())
    file.close()

app = wx.App()
win = wx.Frame(None, title="Simple Editor", size=(410, 335))

bkg = wx.Panel(win)

loadButton = wx.Button(bkg, label="open")
loadButton.Bind(wx.EVT_BUTTON, load)
saveButton = wx.Button(bkg, label="Save")
saveButton.Bind(wx.EVT_BUTTON, save)

filename = wx.TextCtrl(bkg)
contents = wx.TextCtrl(bkg, style=wx.TE_MULTILINE | wx.HSCROLL)

hbox = wx.BoxSizer()
hbox.Add(filename, proportion=1, flag=wx.EXPAND)
hbox.Add(loadButton, proportion=0, flag=wx.LEFT, border=5)
hbox.Add(saveButton, proportion=0, flag=wx.LEFT, border=5)

vbox = wx.BoxSizer(wx.VERTICAL)
vbox.Add(hbox, proportion=0, flag=wx.EXPAND | wx.ALL, border=5)
vbox.Add(contents, proportion=1,
         flag=wx.EXPAND | wx.LEFT | wx.BOTTOM | wx.RIGHT, border=5)

bkg.SetSizer(vbox)
win.Show()

app.MainLoop()
```

`GUI.py`

可以按照下面的步骤使用这个编辑器。

(1) 运行程序。应该看到一个和刚才差不多的窗口。

(2) 在文本区里面打些字(比如 `"Hello, world!"`)

(3) 在文本框中键入文件名(比如 `hello.txt`)。确保文件不存在，否则它会被覆盖。

(4) 点击 `Save` 按钮。

(5) 关闭编辑窗口(只为了好玩)。

(6) 重启程序。

(7) 在文本框内键入同样的文件名。

(8) 点击 `Open` 按钮。文件的文本内容应该会在大文本区内重现。

(9) 随便编辑一下文件，再次保存。

现在可以打开、编辑和保存文件，直到感到烦为止——然后应该考虑一下改进。例如使用 `urllib` 模块让程序下载文件怎么样？

读者可能会考虑在程序中使用更加面向对象的设计。例如，可能希望将主应用程序作为自定义应用程序类(可能是 `wx.App` 的子类)的一个实例进行管理，而不是将整个结构置于程序的最顶层。也可以创建一个单独的窗口类(`wx.Frame` 的子类)。请参见第28章获取更多示例。

PYW怎么样

在Windows中，你可以使用 `.pyw` 作为文件名的结尾来保存GUI程序。在第一章中，我告诉给你的文件名使用这个结尾然后双击它(Windows中)，什么都没发生，之后我保证我会在后面解释。在第十章中，我再次提到了这个是，并且说本章内会解释，所以现在就说说吧。

其实没什么大不了的。在Windows中双击普通的Python脚本时，会出现一个带有Python提示符的DOS窗口，如果使用 `print` 和 `raw_input` 作为基础界面，那么就没问题。但是现在已经知道如何创建GUI程序了，DOS窗口就显得有些多余， `pyw` 窗口背后的真相就是它可以在没有DOS窗口的情况下运行Python——对于GUI程序就完美了。

12.4 但是我宁愿用……

Python的GUI工具包是在太多，所以我没办法将所有工具包都展示给你看。不过我可以给出一些流行的GUI包中的例子(比如Tkinter和Jython/Swing)。

为了演示不同的包，我创建了一个简单的程序——很简单，比刚才的编辑器例子还简单。只有一个窗口，该窗口包含一个带有 `"Hello"` 标签的按钮。当点击按钮时，它会打印出文本 `"Hello, world!"` ，为了简单，我没有使用任何特殊的布局特性。下面是一个wxPython版本的示例。

```
import wx

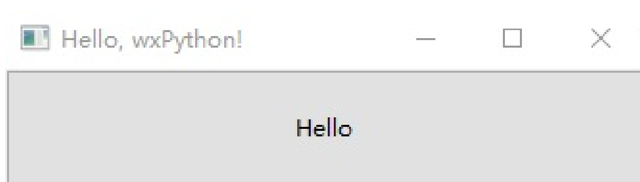
def hello(event):
    print "Hello, world!"
app = wx.App()

win = wx.Frame(None, title="Hello, wxPython!",
               size=(200, 100))
button = wx.Button(win, label="Hello")
button.Bind(wx.EVT_BUTTON, hello)

win.Show()
app.MainLoop()
```

最终的结果如图12-6所示。

图12-6 简单的GUI示例



12.4.1 使用Tkinter

Tkinter是个老牌的Python GUI程序。它由Tk GUI工具包(用于Tcl编程语言)包装而来。默认在Windows版和Mac OS发布版中已经包括。下面的网址可能有用：

☑ http://www.ibm.com/developerworks/linux/library/l-tkprg

☑ http://www.nmt.edu/tcc/help/lang/python/tkinter.pdf

下面是使用Tkinter实现的GUI程序。

```
from Tkinter import *

def hello():
    print "Hello, world!"
# Tkinter的主窗口
win = Tk()
win.title("Hello, Tkinter!")
# Size 200, 100
win.geometry("200x100")
btn = Button(win, text="Hello", command=hello)
btn.pack(expand=YES, fill=BOTH)

mainloop()
```

12.4.2 使用Jython和Swing

如果正在使用Jython(Python的Java实现)，类似wxPython和Tkinter这样的包就不能用了。唯一可用的GUI工具包是Java标准库包AWT和Swing(Swing是最新的，而且被认为是标准的Java GUI工具包)。好消息是两者都直接可用，不用单独安装。更多信息，请访问Java的网

站，以及为Java而写的Swing文档：

☑ http://www.jython.org

☑ http://java.sun.com/docs/books/tutorial/uiswing

下面是使用Jython和Swing实现的GUI示例。

```
from javax.swing import *
import sys
def hello(event):
    print "Hello, world!"
btn = JButton("Hello")
btn.actionPerformed = hello

win = JFrame("Hello, Swing!")
win.contentPane.add(btn)
def closeHandler(event):
    sys.exit()

win.windowClosing = closeHandler

btn.size = win.size = 200, 100

win.show()
```

注意，这里增加了一个额外的事件处理函数(`closeHandler`)，因为关闭按钮Java Swing中没有任何有用的默认行为。另外，无须显式地进入主事件循环，因为它是和程序并行运行的(在不同的线程中)。

12.4.3 使用其他开发包

大多数GUI工具包的基础都一样，不过遗憾的是当学习如何使用一个新的包时，通过使你能做成想做的事情的所有细节而找到学习新包的方法还是很花时间的。所以在决定使用哪个包(12.1节应该对从何处着手有些帮助)之前应该花上些时间考虑，然后就是泡在文档中，写代码。希望本章能提供了那些理解文档时需要的基础概念。

12.5 小结

再来回顾一下本章讲了什么。

☑ 图形用户界面(GUI)：GUI可以让程序更友好。虽然并不是所有的程序都需要它们，但是当程序要和用户交互时，GUI可能会有所帮助。

☑ Python的GUI平台：Python程序员有很多GUI平台可用。尽管有这么多选择是好事，但是选择时有时会很困难。

☑ wxPython：wxPython是成熟的并且特色丰富的跨平台的Python GUI工具包。

☑ 布局：通过指定几何坐标，可以直接将组件放置在想要的位置。但是，为了在包含它们的窗口改变大小时能做出适当的改变，需要使用布局管理器。wxPython中的布局机制是尺寸器。

☑ 事件处理：用户的动作触发GUI工具包的事件。任何应用中，程序都会有对这些事件的反应，否则用户就没法和程序交互了。wxPython中事件处理函数使用`Bind`方法添加到组件上。

接下来学什么

本章内容就是这样了。已经学习完了如何编写通过文件和GUI与外部世界交互的程序。下一章将会介绍另外一个很多程序系统都具有的重要组件：数据库。

第十三章 数据库支持

> 来源：http://www.cnblogs.com/Marlowes/p/5537223.html
>
> 作者：Marlowes

使用简单的纯文本文件只能实现有限的功能。没错，使用它们可以做很多事情，但有时需要额外的功能。你可能想要自动序列化，这时可以选择 `shelve` 模块(见第十章)和 `pickle` (与 `shelve` 模块关系密切)。但有时，可能需要比这更强大的特性。例如，可能想自动地支持数据并发访问——想让几个用户同时对基于磁盘的数据进行读写而不造成任何文件损坏这类的问题。或者希望同时使用多个数据字段或属性进行复杂的搜索，而不是只通过 `shelve` 做简单的单键查找。解决的方案有很多，但是如果要处理的数据量巨大而同时还希望其他程序员能轻易理解的话，选择相对来说更标准化的数据库(database)可能是个好主意。

本章会对Python的Database API进行讨论，这是一种连接SQL数据库的标准化方法；同时也会展示如何用API执行一些基本的SQL命令。最后一节会对其他可选的数据库技术进行讨论。

我不打算把这章写成关系型数据库或SQL语言的教程。多数数据库的文档(比如PostgreSQL、MySQL，以及本章用到的SQLite数据库)都应该能提供相关的基础知识。如果以前没用过关系型数据库，也可以访问 http://www.sqlcourse.com ，或者干脆网上搜一下相关主题，或查看由Clare Churcher著的*Beginning SQL Queries*(Apress，2008年出版)。

当然，本章使用的简单数据库(SQLite)并不是唯一的选择。还有一些流行的商业数据库(比如Oracle或Microsoft SQL Server)以及很多稳定且被广泛使用的开源数据库可供选择(比如MySQL、PostgreSQL和Firebird)。第二十六章中使用了PostgreSQL，并且介绍了一些MySQL和SQLite的使用指导。关于其他Python包支持的数据库，请访问 http://www.python.org/topics/database/ ，或者访问Vaults of Parnassus的数据库分类。

关系型(SQL)数据库不是唯一的数据库类别。还有不少类似于ZODB的对象数据库、类似Metakit基于表的精简数据库，和类似于BSD DB的更简单的键-值数据库。

本章着重介绍低级数据库的交互，你会发现几个高级库可以帮助完成一些复杂的工作(例如，参见 http://www.sqlalchemy.org 或者 http://www.sqlobject.org ，或者在网络上搜索Python的对象-关系映射)。

13.1 Python数据库API

支持SQL标准的可用数据库有很多，其中多数在Python中都有对应的客户端模块(有些数据库甚至有多个模块)。所有数据库的大多数基本功能都是相同的，所以写一个程序来使用其中的某个数据库是很容易的事情，而且“理论上”该程序也应该能在别的数据库上运行。在提供相同

功能(基本相同)的不同模块之间进行切换时的问题通常是它们的接口(API)不同。为了解决Python中各种数据库模块间的兼容问题，现在已经通过了一个标准的DB API。目前的API版本(2.0)定义在PEP249中的Python Database API Specification v2.0中。

本节将对基本概念做一综述。并且不会提到API的可选部分，因为它们不见得对所有数据库都适用。可以在PEP中找到更多的信息，或者可以访问官方的Python维基百科中的数据库编程指南。如果对API的细节不感兴趣，可以跳过本节。

13.1.1 全局变量

任何支持2.0版本DB API的数据库模块都必须定义3个描述模块特性的全局变量。这样做的原因是API设计得很灵活，以支持不同的基础机制、避免过多包装，可如果想让程序同时应用于几个数据库，那可是件麻烦事了，因为需要考虑到各种可能出现的状况。多数情况下，比较现实的做法是检查这些变量，看看给定的数据库模块是否能被程序接受。如果不能，就显示合适的错误信息然后退出，例如抛出一些异常。3种全局变量如表13-1所示。

表13-1 Python DB API的模块特性

```
apilevel                所使用的Python DB API版
threadsafety             模块的线程安全等级
paramstyle               在SQL查询中使用的参数风格
```

API级别(`apilevel`)是个字符串常量，提供正在使用的API版本号。对DBAPI 2.0版本来说，其值可能是'1.0'也可能是'2.0'。如果这个变量不存在，那么模块就不适用于2.0版本，根据API应该假定当前使用的是DB API 1.0。在程序中提供对其他可能值的支持没有坏处，谁知道呢，说不定什么时候DBAPI的3.0版本就出来了。

线程安全性等级(`threadsafety`)是个取值范围为0~3的整数。0表示线程完全不共享模块，而3表示模块是完全线程安全的。1表示线程本身可以共享模块，但不对连接共享(参见13.1.3节)。如果不使用多个线程(多数情况下可能不会这样做)，那么完全不用担心这个变量。

参数风格(`paramstyle`)表示在执行多次类似查询的时候，参数是如何被拼接到SQL查询中的。值`'format'`表示标准的字符串格式化(使用基本的格式代码)，可以在参数中进行拼接的地方插入`%s`。而值`'pyformat'`表示扩展的格式代码，用于字典拼接中，比如`%(foo)`。除了Python风格之外，还有第三种接合方式：`'qmark'`的意思是使用问号，而`'numeric'`表示使用`:1`或者`:2`格式的字段(数字表示参数的序号)，而`'named'`表示`:foobar`这样的字段，其中`foobar`为参数名。如果参数风格看起来有些让人迷惑，别担心。对于基础程序来说，不会用到这些参数，如果需要了解特定的数据库接口如何处理参数，在相关的文档中会进行解释。

13.1.2 异常

为了能尽可能准确地处理错误，API中定义了一些异常类。它们被定义在一种层次结构中，所以可能通过一个`except`块捕捉多种异常。(当然要是你觉得一切都能运行良好，或者根本不在乎程序因为某些事情出错这类不太可能发生的时间而突然停止运行，那么完全可以忽略这些异常)

异常的层次如表13-2所示。在给定的数据库模块中异常应该是全局可用的。关于这些异常的深度描述，请参见API规范(也就是前面提到的PEP)。

表13-2 在DB API中使用的异常

```
异常                    超类                    描述
StandardError                                     所有异常的泛型基类
Warning                  StandardError             在非致命错误发生时引发
Error                   StandardError             所有错误条件的泛型超类
InterfaceError            Error                   关于接口而非数据库的错误
DatabaseError             Error                   与数据库相关的错误的基类
DataError                DatabaseError             与数据库相关的问题，比如值超出范围
OperationalError          DatabaseError             数据库内部操作错误
IntegrityError            DatabaseError             关系完整性受到影响，比如键检查失败
InternalError             DatabaseError             数据库内部错误，比如非法游标
ProgrammingError          DatabaseError             用户编程错误，比如未找到表
NotSupportedError          DatabaseError             请求不支持的特性，比如回滚
```

13.1.3 连接和游标

为了使用基础数据库系统，首先必须连接到它。这个时候需要使用具有恰当名称的`connect`函数，该函数有多个参数，而具体使用哪个参数取决于数据库。API定义了表13-3中的参数作为准则，推荐将这些参数作为关键字参数使用，并按表中给定的顺序传递它们。参数类型都应为字符串。

表13-3 connect函数的常用参数

```
参数名                   描述                                 是否可选
dsn                      数据源名称，给出该参数表示数据库依赖       否
user                     用户名                                是
password                  用户密码                               是
host                     主机名                                是
database                  数据库名                               是
```

13.2.1节以及第二十六章会介绍使用`connect`函数的具体的例子。

`connect`函数返回连接对象。这个对象表示目前和数据库的会话。连接对象支持的方法如表13-4所示。

13-4 连接对象方法

```
close()                                  关闭连接之后，连接对象和它的游标均不可用
commit()                                 如果支持的话就提交挂事务，否则不做任何事
rollback()                                回滚挂起的事务(可能不可用)
cursor()                                 返回连接的游标对象
```

`rollback` 方法可能不可用，因为不是所有的数据库都支持事务(事务是一系列动作)。如果可用，那么它就可以“撤销”所有未提交的事务。

`commit` 方法总是可用的，但是如果数据库不支持事务，它就没有任何作用。如果关闭了连接但还有未提交的事务，它们会隐式地回滚——但是只有在数据库支持回滚的时候才可以。所以如果不想完全依靠隐式回滚，就应该每次在关闭连接前进行提交。如果提交了，那么就用不着担心关闭连接的问题，它会在进行垃圾收集时自动关闭。当然如果希望更安全一些，就调用 `close` 方法，也不会敲很多次键盘。

`cursor` 方法将我们引入另外一个主题：游标对象。通过游标执行SQL查询并检查结果。游标比连接支持更多的方法，而且可能在程序中更好用。表13-5给出了游标方法的概述，表13-6则是特性的概述。

表13-5 游标对象方法

```
callproc(name[, params])       使用给定的名称和参数(可选)调用已命名的数据库程序
close()                        关闭游标之后，游标不可用
execute(oper[, params])        执行SQL操作，可能使用参数
executemany(oper, pseq)        对序列中的每个参数执行SQL操作
fetchone()                     把查询的结果集中的下一行保存为序列，或者None
fetchmany([size])              获取查询结果集中的多行，默认尺寸为arraysize
fetchall()                     将所有(剩余)的行作为序列的序列
nextset()                      跳至下一个可用的结果集(可选)
setinputsizes(sizes)           为参数预先定义内存区域
setoutputsize(size[, col])     为获取的大数据值设定缓冲区尺寸
```

表13-6 游标对象特性

```
description                    结果列描述的序列，只读
rowcount                       结果中的行数，只读
arraysize                      fetchmany中返回的行数，默认为1
```

其中一些方法会在下面详细介绍，而有些(比如 `setinputsizes` 和 `setoutputsizes`)则不会提到。更多细节请查阅PEP。

13.1.4 类型

数据库对插入到具有某种类型的列中的值有不同的要求，是为了能正确地与基础SQL数据库进行交互操作，DB API定义了用于特殊类型和值的构造函数以及常量(单例模式)。例如，如果想要在数据库中增加日期，它应该用相应的数据库连接模块的 `Date` 构造函数来建立。这样数据库连接模块就可以在幕后执行一些必要的转换操作。所有模块都要求实现表13-7中列出的构造函数和特殊值。一些模块可能不是完全按照要求去做，例如 `sqlite3` 模块(接下来会讨论)并不会输出表13-7中的特殊值(通过 `ROWIP` 输出 `STRING`)。

表13-7 DB API构造函数和特殊值

```
Date(year, month, day)              创建保存日期值的对象
Time(hour, minute, second)          创建保存时间值的对象
Timestamp(y, mon, d, h, min, s)     创建保存时间戳值的对象
DateFromTicks(ticks)                创建保存自新纪元以来秒数的对象
TimeFromTicks(ticks)                创建保存来自秒数的时间值的对象
TimestampTicks(ticks)               创建保存来自秒数的时间戳的对象
Binay(string)                       创建保存二进制字符串值的对象
STRING                              描述基于字符串的列类型(比如CHAR)
BINARY                              描述二进制列(比如LONG或RAW)
NUMBER                              描述数字列
DATETIME                            描述日期/时间列
ROWID                               描述行ID列
```

13.2 SQLite和PySQLite

之前提到过，可用的SQL数据库引擎有很多，而且都有相应的Python模块。多数数据库引擎都作为服务器程序运行，连安装都需要管理员权限。为了降低练习Python DB API的门槛，这里选择了小型的数据库引擎SQLite，它并不需要作为独立的服务器运行，并且不基于集中式数据库存储机制，而是直接作用于本地文件。

在最近的Python版本中(从2.5开始)，SQLite的优势在于它的一个包装(PySQLite)已经被包括在标准库内。除非是从源码开始编译Python，可能数据库本身也已经包括在内。读者也可以尝试13.2.1节介绍的程序段。如果它们可以工作，那么就不用单独安装PySQLite和SQLite了。

注：如果读者没有使用*PySQLite*的标准库版本，那么可能还需要修改`import`语句，请参考相关文档获取更多信息。

获取**PySQLite**

如果读者正在使用旧版Python，那么需要在使用SQLite数据库前安装PySQLite，可以从官方网站下载。对于带有包管理系统的Linux系统，可能直接从包管理器章获得PYSQLite和SQLite。

针对PYSQLite的Windows二进制版本实际上包含了数据库引擎(也就是SQLite)，所以只要下载对应Python版本的PYSQLite安装程序，运行就可以了。

如果使用的不是Windows，而操作系统也没有可以找到PYSQLite和SQLite的包管理器的话，那么就需要PYSQLite和SQLite的源代码包，然后自己进行编译。

如果使用的Python版本较新，那么应该已经包含PySQLite。接下来需要的可能就是数据库本身SQLite了(同样，它可能也包含在内了)。可以从SQLite的网站 http://sqlite.org 下载源代码(确保得到的是已经完成自动代码生成的包)，按照README文件中的指导进行编译即可。在之后编译PYSQLite时，需要确保编译过程可以访问SQLite的库文件和include文件。如果已经在某些标准位置安装了SQLite，那么可能SQLite发布版的安装脚本可以自己找到它，在这种情况下只需执行下面的命令：

```
python setup.py build
python setup.py install
```

可以只用后一个命令，让编译自动进行。如果出现大量错误信息，可能是安装脚本找不到所需文件。确保你知道库文件和`include`文件安装到了哪里，将它们显式地提供给安装脚本。假设我在`/home/mlh/sqlite/current`目录中原地编译SQLite，那么头文件和库文件应该可以在`/home/mlh/sqlite/current/src`和`/home/mlh/sqlite/current/build/lib`中找到。为了让安装程序能使用这些路径，需要编辑安装脚本`setup.py`。在这个文件中可以设定变量`include_dirs`和`library_dirs`：

```
include_dirs = ['/home/mlh/sqlite/current/src']
include_dirs = ['/home/mlh/sqlite/current/build/lib']
```

在重新绑定变量之后，刚才说过的安装过程应该可以正常进行了。

13.2.1 入门

可以将SQLite作为名为`sqlite3`的模块导入(如果使用的是标准库中的模块)。之后就可以创建一个到数据库文件的连接——如果文件不存在就会被创建——通过提供一个文件名(可以是文件的绝对或者相对路径)：

```
>>> import sqlite3
>>> conn =  sqlite3.connect("somedatabase.db")
```

之后就能获得连接的游标：

```
>>> curs = conn.cursor()
```

这个游标可以用来执行SQL查询。完成查询并且做出某些更改后确保已经进行了提交，这样才可以将这些修改真正地保存到文件中：

```
>>> conn.commit()
```

可以(而且是应该)在每次修改数据库后都进行提交，而不是仅仅在准备关闭时才提交，准备关闭数据库时，使用`close`方法：

```
>>> conn.close()
```

13.2.2 数据库应用程序示例

我会建立一个小型营养成分数据库作为示例程序，这个程序基于USDA的营养数据实验室提供的数据。在他们的主页上点击USDA National Nutrient Database for Standard Reference链接，就能看到很多以普通文本形式(ASCII)保存的数据文件，这就是需要的内容。点击Download链接，下载标题"Abbreviated"下方的ASCII链接所指向的ASCII格式的`zip`文件。此时应该得到一个`zip`文件，其中包含`ABBREV.txt`文本文件和描述该文件内容的PDF文件。

`ABBREV.txt`文件中的数据每行都有一个数据记录，字段以脱字符(`^`)进行分割。数字字段直接包含数字，而文本字段包括由波浪号(`~`)括起来的字符串值，下面是一个示例行，为了简短起见删除了一部分：

```
~01252~^~CHEESE ... ,PAST PROCESS, ... ^~1 slice,  (3/4 oz)~^0
```

用`line.split("^")`可以很容易地将这样一行文字解析为多个字段。如果字段以波浪号开始，就能知道它是个字符串，可以用`field.strip("~")`获取它的内容。对于其他的(数字)字段来讲可以使用`float(field`)，除非字段是空的。下面一节中的程序将演示把ASCII文件中的数据移入SQL数据库，然后对其进行一些有意思的查询。

注：这个示例程序有意提供一个简单的例子。有关相对高级的用于*Python*的数据库的例子，参见第二十六章。

1.创建和填充表

为了真正地创建数据库表并且向其中插入数据，写个完全独立的一次性程序可能是最简单的方案。运行一次后就可以忘了它和原始数据源(`ABBREV.txt`文件)，尽管保留它们也是不错的主意。

代码清单13-1中的程序创建了叫做`food`的表和适当的字段，并且从`ABBREV.txt`中读取数据。之后分解析(行分解为多个字段，并使用应用函数`convert`每个字段进行转换)，然后通过调用`curs.execute`执行SQL的`INSERT`语句将文本字段中的值插入到数据库中。

注：也可以使用*`curs.executemany`*，然后提供一个从数据文件中提取的所有行的列表。这样做在本例中只会带来轻微的速度提升，但是如果使用通过网络连接的客户机/服务器*SQL*系统，则会大大地提高速度。

```
import sqlite3
def convert(value):
    if value.startwith("~"):
        return value.strip("~")
    if not value:
        value = 0
        return float(value)

conn = sqlite3.connect("foo.db")
curs = conn.cursor()

curs.execute("""
CREATE TABLE food (
id        TEXT        PRIMARY KEY,
desc      TEXT,
water     FLOAT,
kcal      FLOAT,
protein   FLOAT,
fat       FLOAT,
ash       FLOAT,
carbs     FLOAT,
fiber     FLOAT,
sugar     FLOAT
) """)

query = "INSERT INTO food VALUES (?,?,?,?,?,?,?,?,?,?)"

for line in open("ABBREV.txt"):
    fields = line.split("^")
    vals = [convert(f)
    for f in fields[:field_count]]
        curs.execute(query, vals)

conn.commit()
conn.close()
```

`importdata.py`

注：在代码清单*13-1*中使用了`paramstyle`的“问号”版本，也就是会使用问号作为字段标记。如果使用旧版本的*PySQLite*，那么久需要使用`%`字符。

当运行这个程序(将`ABBREV.txt`放在同一目录)时，它会创建一个叫做`food.db`的新文件，它会包含数据库中的所有数据。

鼓励读者们多尝试修改这个例子，例如使用其他的输入、加入`print`语句，等等。

2.搜索和处理结果

使用数据库很简单。再说一次，需要创建连接并且获得该链接的游标。使用`execute`方法执行SQL查询，用`fetchall`等方法提取结果。代码清单13-2展示了一个将SQL SELECT条件查询作为命令行参数，之后按记录格式打印出返回行的小程序。可以用下面的命令尝试这个程序：

```
$ python food_query.py "kcal <= 100 AND fiber >= 10 ORDER BY sugar"
```

运行的时候可能注意到有个问题。第一行，生橘子皮(raw orange peel)看起来不含任何糖分(糖分值为0)。这是因为在数据文件中这个字段丢失了。可以改进刚才的导入脚本检测条件，然后插入None来代替真实的值来表示丢失的数据。可以使用如下条件：

```
"kcal <= 100 AND fiber >= 10 AND sugar ORDER BY sugar"
```

请求在任何返回行中包含实际数据的“糖分”字段。这方法恰好也适用于当前的数据库，它会忽略糖分为0的行。

注：使用*ID*搜索特殊的食品项，比如用*08323*搜索*Cocoa Pebble*的时候可能会出现问题。原因在于*SQLite*以一种相当不标准的方式处理它的值。在其内部所有的值实际上都是字符串，一些转换和检查在数据库和*Python API*间进行。通常它工作得很顺利，但有时候也会出错，例如下面这种情况：如果提供值*08323*，它会被解释为数字*8323*，再转换为字符串*"8323"*——一个不存在的*ID*。可能期待这里抛出异常或者其他什么错误信息，而不是这种毫无用处的非预期行为。但如果小心一些，一开始就用字符串*"08323"*来表示*ID*，就可以工作正常了。

```
import sqlite3 import sys

conn = sqlite3.connect("foo.db")
curs = conn.cursor()

query = "SELECT * FROM food WHERE %s" % sys.argv[1]
print query

curs.execute(query)
names = [f[0] for f in curs.description]
for row in curs.fetchall():
    for pair in zip(names, row): print "%s: %s" % pair print
```

`food_query.py`

13.3 小结

本章简要介绍了创建和关系型数据库交互的Python程序。这段介绍相当简短，因为掌握了Python和SQL以后，那么两者的结合——Python DB API也就容易掌握了。下面是本章一些概念。

☑ Python DB API：提供了简单、标准化的数据库接口，所有数据库的包装模块都应当遵循这个接口，以易于编写跨数据库的程序。

☑ 连接：连接对象代表的是和SQL数据库的通信连接。使用 `cursor` 方法可以从它那获得独立的游标。通过连接对象还可以提交或者回滚事务。在处理完数据库之后，连接可以被关闭。

☑ 游标：用于执行查询和检查结果。结果行可以一个一个地获得，也可以很多个(或全部)一起获得。

☑ 类型和特殊值：DB API标准制定了一组构造函数和特殊值的名字。构造函数处理日期和时间对象，以及二进制数据对象。特殊值用来表示关系型数据库的类型，比如`STRING`、`NUMBER`和`DATETIME`。

☑ SQLite：小型的嵌入式SQL数据库，它的Python包装叫做PYSQLite。它速度快，易于使用，并且不需要建立单独的服务器。

13.3.1 本章的新函数

本章涉及的新函数如表13-8所示。

表13-8 本章的新函数

```
connect(...)                            连接数据库，返回连接对象
```

13.3.2 接下来学什么

坚持不懈数据库处理是绝大多数程序(如果不是大多数，那就是大型程序系统)的重要部分。下一章会介绍另外一个大型程序系统都会用到的组件，即网络。